コンピュータ・サイエンス研究書シリーズ ㉞

オペレーティング・システムの基礎

トロント大学教授 Tsichritzis/Bernstein著
慶応義塾大学工学部助教授 斎藤信男訳

日本コンピュータ協会

コンピュータ・サイエンス研究書シリーズ ㉞

オペレーティング・システムの基礎

トロント大学教授 Tsichritzis/Bernstein著
筑波大学電子・情報工学系助教授 斎藤信男訳

日本コンピュータ協会

OPERATING SYSTEMS

Dionysios C. Tsichritzis and Philip A. Bernstein
University of Toronto

訳 者 序 文

本書は，オペレーティング・システムの基礎的な知識を解説することを目的として書かれている．オペレーティング・システムの詳細な各論はむしろ意識的に省いて，オペレーティング・システムで用いられている原理，概念，基本的技法を懇切丁寧に説明している．

計算機を利用して仕事をしようとする人にとっては，オペレーティング・システムは，直接係わりあいのあるソフトウェア・システムである．これを正しく理解していなければ，計算機の利用を効率よく行なうことができない．しかし，オペレーティング・システムは，機械に依存して設計されているので，個々の機械のシステムの内容を詳細に知るためには，かなりの時間と労力とを要する．そこで，最も能率のよい方法は，オペレーティング・システムで用いられている基本的な原理や概念をきちんと理解し，それらと現実のシステムとの対応をつけながら，理解してゆくという方法である．このような方法でオペレーティング・システムを理解しておけば，その利用に際して問題が生じたとき，適確に判断することができるわけである．

オペレーティング・システムは，また，計算機科学の研究対象としても非常に興味深いものである．その場合には，オペレーティング・システムの動作原理や基本構造を正しく理解することが研究の第一歩となる．

オペレーティング・システムの基礎知識は，上述の意味で，情報工学関係の学部と大学院の学生達あるいは計算機を頻繁に利用する可能性のある学生達にとって必須のことである．本書は，そのような基礎知識を習得する際に大いに役立つことをねらっている．いくつかあるオペレーティング・システムの参考書の中で，そのような目的をもって書かれたものは，例を見ない．

著者 Tsichritzis 教授は，トロント大学の SUE プロジェクトの発案者の一

人であり，同大学計算機科学科の中心メンバーとして活躍されている．本書は，同教授が行なったオペレーティング・システムの講義の記録に基づいて書かれており，その内容にはSUEプロジェクトの参加者の考えや研究成果が生かされている．かんで含めるような丁寧な解説には，著者の人柄がにじみ出ている．同教授は，また，COSINE作業部会のメンバーであり，オペレーティング・システムの教育カリキュラムの編成に多くの影響を与えている．本書は，COSINE作業部会のメンバーの著書としては最初のものであり，同作業部会の報告に沿った**最も標準的な教科書**である．なお，COSINE作業部会のメンバーの著書として，P. J. Denning教授の"Operating Systems Theory"（訳書：**オペレーティング・システムの理論，**日本コンピュータ協会）があるが，その翻訳の出版が計画されており，大いに期待されている．

"講義担当者への注意"の項や"おもちゃのオペレーティング・システム"の項を読むと，教室での講義以外に補助教材を用いた演習がどのように行なわれているのかを知ることができて，興味深い．計算機に関する講義は，このように"生きた"講義として実施されなければならないが，そのためには大変な努力が必要であることを痛感させられる．

設計と実現についてはそれぞれ章をもうけているが，これだけまとまった解説は他書に例を見ない．最近話題になっているソフトウェア工学と関連づけて読めば，参考になるであろう．

訳語に関しては，なるべく日本語にするように心がけたが，片かなであらわすことが既に定着しているもの，まだ適当な日本語の決っていないものは，片かなを用いてあらわした．また，重要な用語はその原語がわかるように，最初にあらわれたときに原語をつけておいた．

本書の翻訳に際して，終始協力して頂いた**日本コンピュータ協会**の村越勝弘氏に感謝の意を捧げる．また，索引作り，校正などを手伝ってくれた妻晴子にも感謝する次第である．

1976年3月

斎藤信男

まえがき

オペレーティング・システムとは何であるかを，正確に定義することは難しい．“システム”という術語は，計算機科学や他の工学の分野において，いろいろの意味に用いられているが，“オペレーティング”という言葉を付加しても，その意味はたいして明瞭にはなってこない．歴史的に考えれば，計算機のユーザは，そのハードウェアが与えてくれるものとは異った機能を，システムに期待してきた．この問題を解決するために，通常，幾つかのプログラムを用意し，それらがハードウェアの機能を変換して，ユーザの要求に合うような機能を提供しているのである．この一群のプログラムのことを**オペレーティング・システム**（operating system）と呼んでいる．ここでは，この術語をこれ以上詳しく定義したり，混乱させたりすることはしないでおこう．第1章において，オペレーティング・システムとは何から構成されているのかという点に関して，われわれの見解を提示している．出来ることならば，本書の終りまで読んだ読者は，オペレーティング・システムというものをきちんと把握できるようになって欲しいものである．

大部分のユーザは，自分のプログラムを処理しているオペレーティング・システムを，完全に理解している必要はない．メモリの割り当ての大きさや，ジョブ制御言語などのようなシステムから与えられる制限さえ守っていればよいのである．同様にして，FORTRAN のプログラマは，自分が用いている FORTRAN のコンパイラの細部まで，すべて知っている必要はない．しかしながら，オペレーティング・システムの原理にふれてみることが，“計算機科学者”にとって必要な経験であることは，誰でも認めることであろう．本書の目的はこのような必須の知識を読者に与えることである．オペレーティング・システムを設計する一般的な方法を定義するつもりはない．むしろ，オペレーティン

グ・システムの動作を規定する基本的な概念や原理を，簡潔で明瞭な方法で提示することを試みている．

ここで用いる用語は，出来る限り文献等と一貫性を保つようにしている．ハードウェアに関しては，計算機メーカがしばしば矛盾した定義を用いるので，一貫性を保つのは困難であった．問題が生じたときは，多数のユーザに使われているという事実を考慮して，IBM の用語を採用した．アルゴリズムを記述する際には，適切な制御構造や基本演算子を勝手に使用できるということを重視して，特に決まったプログラミング言語は用いなかった．たとえば，PASCAL のような適切な言語があるのにも拘わらず，それが広く知られていないために，広く使われている PL/I のような言語を用いて記述するということは好ましいことではない．用語の持つニュアンスは，それらを導入する節において，明確に定義している．

本書は，**ACMカリキュラム68**のコースI4と同じように，学部４年生および大学院１年生を対象とした一学期間の講義の教科書として使うことを主たる目的としている．同様の講義を，1969年以来トロント大学において持っており毎年約100人の受講者がいる．その講義（および本書）は，オペレーティング・システムに関する COSINE† 作業部分［Denning ほか，1971］の勧告に沿う様に作り上げてきた．オペレーティング・システムに関する講義を開設する計画のある講義担当者には，この作業部会の報告書を参考にされる事を強く勧めたい．

われわれの大学でこの講義をとる学生は，**データ構造**および**コンパイラ設計**の講義を聴いてきた人達である．このような背景は絶対に必要というわけではないが，これらの講義はシステム・プログラミングの分野の経験をある程度学生に与えてくれるであろう．本書の対象としている学生の背景については，何か特別な仮定をしているわけではなく，計算機科学に関する経験を少し積んでいることを期待しているだけである．特にいうならば，学生達が何か長いプログラムを書いた経験があるとか，計算機システムとの接点においてあらわれる問題，たとえばジョブ制御言語上の制限事項について苦労することなどを，ある程度理解していることなどは，役に立つであろう．また，読者は，リストと

---

† **訳注．** Computer Science in Engineering

待ち行列についてある程度の知識があることは，やはり重要なことである．この項目については，補遺Iに概略を述べてあるが，これらの背景について弱い学生達に対しては，最初に補講的に教えることも可能であろう．

オペレーティング・システムの構造に関する多くの問題は，良く解明されている．第I部で扱っているプロセス，処理装置，記憶等の管理に関する問題は，しばらくの間，多くの文献が幅広く扱ってきた．未解決の問題は未だ多数残っているけれども，これらの問題はかなりきちんと議論することが可能である．残念ながら，これらの問題はすべて非常にきれいにまとまっているというわけではない．第II部で扱っている問題は，その形式的な取り扱いが未だ初期の段階にあるものである．入出力，ファイル，保全性，保護，設計法，信頼性，性能評価，実現方法等は，すべてオペレーティング・システムに関する重要な側面である．これらの項目に関しては，それぞれの問題に対する個別的な技法をたくさん紹介することにしかならないけれども，それにも拘わらずそれらは重要な事項であり，オペレーティング・システムの原理に対する入門的な提示の中には必ず含まれていなければならないと考えている．

オペレーティング・システムの基本概念のうちのいくつかは，本書の中ではかなり後になってからあらわれてくるが，たとえば，仮想機械の概念は第8章で扱う．われわれの経験から考えると，オペレーティング・システムの多くの事例を比較・検討できるだけの知識を持つようになるまでは，このような概念の利点を理解することを学生に期待するのは無理である．最初に扱ういくつかの項目は，これらの概念を用いるともっときれいに提示することが可能かもしれないが，入門書においてはかえって混乱を招くことになるだろう．

オペレーティング・システムの解析的な問題を扱った論文は，多数存在する．これらの研究は，オペレーティング・システムの動作に対して，しばしば，多くの示唆を与えるような結果を生み出してきた．本書では，2つの理由からこれらの解析的方法の結果の大部分を扱うことをやめている．第一の理由は，これらの結果は，オペレーティング・システムの世界よりも，むしろ応用数学の分野に深く関係しているからである．第二の理由は，この項目は基本概念に重点を置いた入門コースで扱うよりも，むしろオペレーティング・システムのモデル化や解析に関する大学院の独立コースで扱う方がずっと適切であるからで

ある．理論的な研究に関しては，本書全体を通して文献を示すことを心がけている．注釈つき参考文献一覧表は，興味のある読者には役に立つであろう．ただし，一つだけ例外を設けることにした．補遺Ⅱでは，第2章に直接関係している項目である計算構造を簡単に概括している．この項目に対する参考文献の大部分のものは，広範囲の人達に手に入り難い状態なので，興味のある読者に対して簡単な紹介を付け加えて置く方がよいだろうと考えたのである．

謝　　　辞

本書で議論している多くのアイデアは，SUEプロジェクトのグループ，すなわち，J. W. Atwood, B. L. Clark, M. S. Grushcow, R. C. Holt, J. J. Horning, K. C. Sevcik, D. C. Tsichritzis によってなされた研究で得られた成果である．このプロジェクトのメンバーは，本書の扱うべき材料ばかりでなく本書の執筆精神にまで深い影響を与えてくれた．彼等の多大な貢献と絶えざる激励とに対し，心から感謝の念を捧げる．

各章の草稿を編集する際には，本学の学生および教官の多くの方々から多大な御援助を頂いた．最終稿が多くの評価を得ることができるならば，それは彼等の忠告に寄るところが大きい．特に，A. J. Ballard（第8章，第9章および補遺Ⅲ），C. C. Gotlieb（第7章），R. C. Holt（第8章），J. J. Horning（第1章および第2章），R. N. S. Horspool（第4章および演習問題），E. Lazowska（第4章および第5章），J. Metzger（第9章），K. C. Sevcik（第3章，第7章および第10章），F. Tompa（第4章～第6章）には，感謝の意を表わしたい．注釈を付けた参考文献一覧表は，R. Bunt が作成したものに基づいている．また，第10章の Venus オペレーティング・システムの記述に際しては，多くの疑問点を明確にする援助を惜しまなかった B. Liskov に感謝の意を表わしたい．一つ前の草稿に対する B. W. Kernighan からの全体にわたる忠告は，いくつかの不正確な部分を見つけ出し，それを修正するために非常に役に立った．

最終稿を注意深く読んでくれ，技術的見地，教育的見地の両方に対し多くの変更点を示唆してくれたわれわれの友人かつ同僚である J. R. Swenson には，特別の感謝の念を捧げたい．特に，記憶の管理に対する彼の多くのアイデアは，第4章～第6章の構成と内容に大きな影響を与えている．

オペレーティング・システムの原理のいろいろの側面に対して，興味のある，

また示唆に富んだ議論をして頂いた他の多くの同僚達に対しても，いちいち名前を挙げないけれども，深く感謝の意を表わしたい．

最後に，本書を作るにあたりいろいろの段階で事務的な仕事を御援助頂いた M. Oldham, V. Shum および P. Steele にも感謝の意を表わしたい．

講義担当者への注意

通常の講義と並行して，一つの特別なシステムをある程度詳細に述べる補講的な講義を設けることは，講義にでてくるアイデアをどの様にして統合するのかを学生に示すためには非常に役に立つ．第10章では，オペレーティング・システムを2例提示している．Hoare と Perrot の著書［1972］および Sayers の著書［1971］には，もっと多くの例が見られる．この種類の議論は，題材に刺激を与えるためにも学期の初めの頃に行なった方が良いと思われる．大抵の学生達，特に学部の学生達は，現実に動いているオペレーティング・システムに接する機会はほとんどないであろう．自分で真価のわからない問題を熱心に解くようになることは，ほとんど期待できないであろう．

各章の終りに付けた演習問題は，本書で扱っている題材を補足しており，多くの有効な目的に使用される．第一に，本文の中では簡単にしか扱っていない項目に対して，詳細な説明を与えてくれるだろう．第二に，幾つかの抽象的な概念を，オペレーティング・システムの実際の世界と関係づけるのに役立つであろう．第三に，さらに勉強するために文献の中の論文を示してくれるだろう．これらの問題の多くは，当然，閉じているものではない．すなわち，正解が唯一つ存在するものではない．そういう種類の問題は，学生達がいろいろの選択を考える事を促すために用意している．これらの問題は，議論の題材として，また，宿題の題材として用いることができる．

トロント大学で開かれている講義で必要とされるものは，次の通りである：

1. 学生達が講義を理解しているかどうかを確かめるための教室における試験．
2. 学生達が理解を深めるための，より本質的な思考力を助長するような問題から成る膨大な宿題．

3. 学生達が，実際に存在するオペレーティング・システムにじかに接することができるようにするためのプロジェクト．

もし少人数のクラスであれば，試験のかわりに短い宿題を幾つか与えてもよいだろう．試験と宿題のための問題は，各章毎の問題の中から選ぶこともできる．

われわれは，プロジェクトに特に力を注ぐだろう．何故ならば，それは講義担当者および学生の双方に対して，きびしい挑戦となるからである．一年間に100 個のプロジェクトを設計し，監督し，評価することは，そんなに易しいことではない．われわれはいくつかの方法を試みてみた．

1. オペレーティング・システムのいくつかの側面を概括した小論文． 問題群の中には，小論文の題材を幾つか提案してある．トロント大学の過去の小論文は集めてあり，将来の学生達に役に立つようにと，図書館に保存してある．コースが開設されてから最初の数年間は，この小論文は特に役に立った．これらは，最小の管理を必要とするだけであり，また，講義の中に含めるのにはあまりに不明確な項目に関して，立派な資料集を身近に作ることができる．特に，世間に良く知られているオペレーティング・システムに関する資料の収集を進めることは，非常に価値のあることである．講義を何回か繰り返した後では，題材を重複させないようにするのが困難となる程であった．この時点まで来れば，他の型のプロジェクトに重点を置くことができるであろう．しかし，ある種の学生達，たとえば，良いプログラマではあるが，文献にはあまり接する機会がないようなパート・タイムの学生達にとっては，小論文のプロジェクトは，1つの選択の余地として残しておくべきである．

2. オペレーティング・システムの設計や実現に関する幾つかの側面を評価するための総合的研究． 最初に W. Wulf によって提案されたシステム・プログラム用言語の評価を行なう Rosetta Stone プロジェクトは，この種の研究の1つの例である [Wulf ほか，1972]．プロセスの同期や記憶管理のアルゴリズムなどの標準的な問題群は，多くの学生達が，二，三の言語を用いてプログラムを作製している．プログラムの作製のほかに異った言語の比較をして意見を述べることを学生達に要求している．この型のプロジェクトを試みたが，

ある程度までしか成功しなかった．主たる原因は，言語を評価するためにはかなりの経験が必要だということである．大部分の学生達は評価を熱心にはしなかったので，新しい言語を二，三覚え，小さな問題をいくつか解いたということが，このプロジェクトの主たる目的となっている．このプロジェクトは，やる価値があるには違いないが，オペレーティング・システムの講義よりも，むしろ，ソフトウェア工学あるいはシステム・プログラミングの講義の方に適しているのだろうと考えている．

3. **シミュレートしたハードウェアの環境を用いた教育用の“おもちゃ”のオペレーティング・システム．**　補遺Ⅲには，このようなプロジェクトを設計し，監督するための基本的問題について概説してある．2～3人のチームを組んだ学生達は，簡単な機械をシミュレートし，その機械に対する小さなオペレーティング・システムを作製する．このプロジェクトは繰り返し試みたが，大成功であった．計算機時間に関しては，少々ぜいたくかもしれないが，学生達は複雑なシステムがどの様に構成されてゆくのかということを，ある程度まで実際に経験できる．側面的効果としては，チームの中でかなり大規模なプログラムを作製するので，学生達はプログラミングに関する実際的な問題を味わうことができる．彼等はプロジェクトの管理の問題も処理しなければならないが，これは，多くのオペレーティング・システムの設計においてやっかいな問題であった．トロント大学では，R. C. Holt が設計した TOPPS プログラミング言語［Holt, Kinread, 1972］を用いたおもちゃのオペレーティング・システムの課題を，現在使用している．TOPPS システム［Czarnik ほか，1973］は，XPL コンパイラ作製システム［Mckeeman ほか，1970］を動かすことのできるシステムの下で，実行することができる．配布のできるテープと資料を近い将来に作って，このプロジェクトを試みようとする大学に使用して貰うことができるようになることを願っている．

4. **ソフトウェアの実験室の環境において，ミニコンピュータを用いたプログラミングのプロジェクト．**　この型のプロジェクトは，非常に面白いものができるであろう．しかし，そのためには，頻繁に使用できるミニコンピュータ・システムが必要となる．もしも適当な設備が使用できるならば，簡単なエ

グゼクティブ（executive），スプーリング・システム（spooling system），簡単なファイル・システムなどいくつかの興味深いソフトウェア・パッケージが一学期のプロジェクトの適当な課題になるであろう．このような環境に関しては，Corbin 等［1971］，Marshland と Tartar［1973］などの報告がいくつかある．当大学においては，この方式はオペレーティング・システムに対する特論のコースで採用している．ただしこの型のプロジェクトを多数監督することは，かなり問題が多くなるであろう．

5.　大学の計算センターあるいは企業に対して，実際に貢献をすること．
われわれは，当大学の計算センターと協定して，計算センターの職員が提示し，監督し，評価し，最終的には実用化して貰うための課題のために，10～15人の学生達を向けている．計算センター側も，学生側も，共に非常に熱心であるが，参加する人員を増やすためには，機構上の重要な困難が生じてくるのではないかと懸念している．これを成功させるためには，各学生はきびしく監督しなければならないが，参加してくれる監督者の数は限られてしまう．このプロジェクトは，実際の経験の少ない学生には特に適している．企業での経験のある学生達は，一般にはシステム・プログラムを，もう一つ作製したいとは思わないのである．経験の少ない学生にとっては，実用的なプログラミングの課題は興味深いものであるだろうが，全く同じ理由で，経験の深いシステム・プログラマにとっては，概括的な小論文を書くことは非常に役に立つであろう．

ある学校において，どのプロジェクトが最も適しているのかを決定する最大の要因は，どれだけの設備が使用できるかということである．前述したプロジェクトの内容について何か疑問の点があるならば，われわれは出来る限りお手伝いすることにやぶさかではない．

目　次

第 I 部

O. S. の 原 理

本書の最初の５章は，計算機システムの基本的な資源，すなわち，処理装置と記憶の割り当てについて論じている．第１章では，オペレーティング・システムが何をなし，それらが通常どのように構成されているかについて，一般的な見解を述べている．第２章では，計算機システムにおいて実行されている計算の間の相互作用を理解する手段として**プロセス**（process）という概念を導入する．第３章では，プロセスを用いて単一の処理装置が幾つかの独立な計算に対してどのように割り当てられるのかを示す．第４章と第５章では，主記憶の割り当てを扱い，また，周辺記憶装置を用いて主記憶を拡張する方法について述べる．

プロセス，処理装置，記憶の管理の方法に関しては，多くの文献が取り上げてきた．その基本的な問題はよく整理されている．したがって，これらの項目を，技術的には正確に，また，教育的には明確に提示することが重要である．

第 1 章

オペレーティング・システムの機能と概念

1.1 はじめに

現代の計算機システムは，“扱いにくい猛獣”と言うことができる．典型的な中規模のシステムの構成は，中央処理装置，コア記憶，ドラム記憶，ディスク装置，テープ装置，タイプライタ端末，カード・リーダ，ライン・プリンタなどの多くのハードウェアの装置が複雑に結合されたものである．このような装置を管理する煩雑さに対処するために，ハードウェア・システムを便利に使用することができるようなプログラムあるいは**ソフトウェア**（software）を，システムが提供している．計算機システムのハードウェアを用いて，プログラムを実行しようと思っている人々，すなわちユーザは，各装置の特殊性を処理するために，ソフトウェアの支援を受ける．彼等は「このプログラムを実行せよ.」とか，「このファイルを格納せよ.」というような操作を要求する．このとき，システム用のソフトウェアは，記憶と処理装置をプログラムに割り当てること，またディスクの空間をファイルに割り当てることなどの細部を管理している．機械を簡単に使えるようにするだけでなく，ソフトウェアは共有することを実現する必要がある．現代の計算機システムは非常に高価なものである．平均的なユーザは，自分自身のシステムを購入することや，大規模なシステムを，単独に専有するための費用を支払うことは不可能であろう．したがって，経済的な必要性から，ユーザは他のユーザとシステムを共有するようにしなければならない．その場合にも，共有が可能となるようにソフトウェアに頼ってシステムの資源を管理しているのである．

この章は，オペレーティング・システムと呼んでいる計算機システムのソフ

トウェアがどんな機能を持ち，どんな性能を要求されているのかという点に関して，読者に概観を与えることが目的である．しかし，オペレーティング・システムを主題とする議論に絞ってしまう前に，計算機システムの構成要素，すなわち，ハードウェアとソフトウェアとの基本的な性質を理解しておくことは重要である．

1.1.1　ハードウェア

計算機システムのハードウェアの構成は，次の通りである．すなわち，機械語の命令を実行する1個または複数個の**中央処理装置**（central processing unit）［**CPU** または **処理装置**（processor）とも呼ぶ］，処理装置が番地付けをできる**記憶**（memory）［**主記憶**（main memory）または**コア**（core）とも呼ぶ］，システムに対する入出力（I/O）の手段を与え，また**二次記憶**（secondary storage）空間を与える**周辺機器**（peripheral devices）［**周辺装置**（peripherals）または**入出力機器**（I/O devices）とも呼ぶ］である．処理装置は，通常，記憶として用いる固定長の**レジスタ**（register）数個と，命令を実行する**演算装置**（arithmetic unit）とから構成されている．主記憶は，**語**（word）と呼ばれる固定長のビットの群に分割されているが，その長さは，通常，レジスタの長さと同じである．記憶内の各語は，**番地**（address）または**位置**（location）と呼ばれる一意的な名前を持っている．処理装置の持っているレジスタの1つは，**命令番地レジスタ**（instruction address register）であるが，それは演算装置が解釈する次の命令の位置を示している．命令は，通常，番地が増加する順序に従って，順次的に実行されてゆくが，**分岐**（branch）の命令まで来ると，命令番地レジスタの内容を変更して新しい位置に分岐する．計算機システムのアーキテクチャ（architecture）は，全く異った構成をとることができるということは，指摘しておかなければならない［Organick, 1973］．

処理装置と記憶に関するこれまでの記述は，実際の場面からすればハードウェアを大幅に単純化したものではあるが，装置に対して何か述べるときの共通の用語を提示している．処理装置と主記憶の性質は，よく知られているだろうが，周辺機器の性質は多数の読者にとっては親しみが薄いと思われるので，それをもう少し詳細に述べてみよう．

周辺機器は，ある種の記憶媒体または表示媒体，たとえば，穿孔カード，磁気ディスクやテープ，プリンタ用紙などと，コアとの間に情報を転送する．各機器は，情報が蓄えられる媒体，情報がアクセスされる方式と速度とが異っている．磁気テープ装置やカード・リーダのような機器は，完全に順次的なものである．データは，**先入先出**（first-in-first-out；略して **FIFO**）の順序に従う単一の流れとしてしか処理できない．たとえば，10番目のデータ項目を読むためには，最初の9個の項目も読まなければならない．他の記憶装置は，データの**乱アクセス**（random access）を許している．磁気ドラムとディスクは，トラックの集合体として構成されている．テープの様に，N番目の項目としてデータの一つにアクセスするのではなく，ドラムではM番目のトラックのN番目の項目としてアクセスする．ドラムやディスクのヘッドはトラックからトラックにかなりの速さで切り換えることができるので，データ項目にアクセスするのに必要な平均時間［すなわち，**アクセス時間**（access time）］は，厳密に順次的に行なう場合比べて，かなり小さくなる．すべての磁気的な機器においては，いったん位置が得られた一つのデータを転送する率［**転送率**（transfer rate）と呼ぶ］は，それを捜し出すためのアクセス時間に比べて，本質的に速い．このような機器の性質から，情報を大きな**ブロック**（block）として転送した方が有利になる．たとえば，10語の情報がディスクに順次的に格納されている場合，これを1回に1語づつアクセスするよりも，ブロック化して1回のアクセスで検索した方がずっと速くできる．小さいブロックのデータでは，アクセス時間に比べて，転送時間は無視できる．したがって，1回に1語づつ転送すれば，10語すべてを1回に転送するのに比べて，おそらく100倍も遅くなるであろう．

機器の種類によって速度はいろいろと異なるけれども，それらを計算機システムに接続する方式はかなり画一的である．周辺機器は，通常，**チャネル**（channel）と呼ぶ特殊目的の計算機を介して CPU と結ぶ．周辺機器からチャネルへの結合は，**機器制御装置**（device control unit）を介して行なわれる．機器制御装置はチャネルから**コマンド**（command）を受け取り，機器をある方法で操作する（たとえば，後戻りとかトラックの切り換えなど）か，または，チャネルへあるいはチャネルからデータを転送する（すなわち，読む，書く）

ことなどを行なうような電子装置である．プログラムが入出力操作を始めると，システムはその要求を受け必要なプログラムを作り出し，それをチャネルへ送る．チャネルは，入出力プログラムに書かれている情報を用いて機器を選び，適当な操作，たとえば，記憶媒体に読み込んだり，書き出したりすることを行なう．入出力操作が終了すると，チャネルは，システムにその終了を知らせなければならない．これは，ハードウェアの割り込みによって知らせる．

割り込み（interrupt）とは，何らかの外的な条件により，実行しているプログラムから強制的に制御を移してしまうことである．割込みの発生に対する要求が認められるときには，**割り込み可能**（enabled）であるという．割り込みの要求および割り込みを可能とさせることは，一般的には，処理装置の特別な割り込みビットを設定することによって行なう．割り込みが要求されかつ割り込み可能であるとき，中央処理装置はそれまで実行していたことを停止し，主記憶のあらかじめ決められた位置へ分岐する．この位置から開始して，割り込みレジスタを調べて割り込みの原因を確かめる．たとえば，入出力機器の処理が終了したとか，不当な命令を実行したとかいう理由で，割り込みが発生したのであろう．そこでチャネルに対し別のコマンドを送るとか，プログラムを消滅させるとか，適当な処置が可能となる．

チャネルと中央処理装置とは，並行的に実行できる．大抵の機械では，CPUのプログラムがチャネルに仕事を指示する責任があるので，独立した機器の操作の同期をとるために，割り込みを用いている．第2章に示すように，割り込みは並行操作の同期をとるのに，必ずしも最適な方法ではない．しかしながら，どのハードウェアにもこの機構が含まれているので，システム設計を理解する上で重要なものとなっている．

1.1.2　ソフトウェア

もしも，ハードウェアが，計算機システムのユーザが必要とする機能をそのまま備えていたとするならば，システム・ソフトウェアは，存在する必要がなくなるであろう．しかし，ハードウェアの機能は，大抵ユーザにとっては最も便利な形で与えられているわけではない．また，ソフトウェアの助けなしに，システムを安全にかつ効率よくユーザが共有することは，困難なことである．

したがって，ユーザは自分のプログラムの処理を支援するために，システム・ソフトウェアを必要とするのである．

1.1.2.1 共　有

資源（resource）とは，システム内で割り当てることのできるすべての物のことである．ハードウェア資源は，2つのプログラムが1つの資源に同時にアクセスするという意味で共有されることは，ほとんどない．むしろ，いくつかのプログラムが，各々，時間周期のある部分を独占的に制御できるときに，資源が**共有**（share）されているという．たとえば，5つのプログラムが，5秒毎に，各々1秒間実行することを許されているとき，それらは平等に処理装置を共有することができる．プログラムに制御を許す時間の規模は，それぞれの機器によって大きく変るかもしれないけれども，理論上はすべてのハードウェア機器を上述の意味で共有している．たとえば，一つのプログラムは主記憶の一部分を数分間使用することを許されるだろうが，処理装置を制御することは，一度には，1秒間以上は決して許されないだろう．

基本的には，共有を実現するのに2つの方式がある．**分散型**（decentralized）のシステムでは，いくつかのプログラムが資源を共有しようと思い，かつプログラムが“お互の存在を知っている”ならば，資源を相互に渡し合いながら協力することができる．しかし，大抵のシステムでは，ユーザはお互に連絡し合うことにわずらわされたくない．**中央集中型**(centralized) 方式では，システムの資源を平等にかつ能率よく割り当てることを，別のプログラムに任しているので，もはや，プログラムが資源への要求を満すために協力し合うという必要はないのである．プログラムが資源を必要とするならば，“**資源割り当て者**(resource allocator)”に，その要求をすればよいだけである．資源割り当てと資源の共有という考え方は，オペレーティング・システムの中心となる概念である．資源割り当ての主な機能とそれらの実現方法については，後節で更に追及する．

1.1.2.2 ソフトウェアのほかの機能

周辺機器のデータの転送の速度は，CPU がデータの処理をする速度に比べて，圧倒的に遅い．CPU がそのデータを待たないために，CPU が実際に必要となる前に，周辺機器はコアにデータを転送しておくべきである．CPU が

コア内のデータの処理を開始したときには，機器はヘッドを動かしはじめており，CPU が停止する前に，CPU をしばらくの間実行状態に保っておくことができるであろう．この操作のことを，**バッファリング**（buffering）と呼び，機器が前もって情報を格納する場所を**バッファ**（buffer）と呼ぶ．計算機システムにおける一つの共通的なバッファリングの技法を**スプーリング**（spooling）† と呼ぶ．処理装置は，通常，その入力データをカード・リーダから得るが，カード・リーダの速度は，処理装置に比べて何桁も遅い．これを補償するために，カードは記憶上の小さいバッファからディスクへ転送（またはスプール）される．ディスクは，ここでは，CPU とカード・リーダとの間のバッファの役をなしている．カードを実際に読むかわりに，CPU はディスクからカードの写しを読むのである．プリンタに対する出力のバッファリングに関しても，同じ方式が適用できる．データのスプーリングを実現するプログラムは，大抵の複雑な計算機システムで，重要視されている．

計算機システムが提供している機能で，最も頻繁に使用されるものの１つとして，コンパイラ(compiler)，アセンブラ(assembler)，インタープリタ（interpreter）などの言語翻訳の機能がある．これは，システム・プログラムの中では，大体の様子がわかっているものに属していると言ってよい．他の多くのソフトウェアの機能と同じように，言語翻訳システムはそれが実行される計算機システムと相互作用があるに違いない．計算機システムと翻訳システムが接する場所を**インターフェイス**（interface）というが，そこは慎重に設計しなければならない．たとえば，翻訳システムは，ファイル記憶とかタイプライタ端末のような資源を，計算機システムから受け取れなければならない．これは，資源割り当てを管理しているシステムと翻訳システムとが連絡しなければならないことを意味する．同様にして，翻訳システムが機械命令コードを生成したとき，そのコードは，たとえばプログラムの結合のための規則のような計算機システムの約束ごとに従っていなければならない．計算機システムの側から考えると，インターフェイスはいろいろ変化に富んだ翻訳システムと結合できるように，簡潔で融通性に富んだものでなければならない．

---

† 訳注．SPOOL は *S*imultaneous *P*eripheral *O*peration *O*n *L*ine の頭文字を並べたもの．

言語翻訳の分野に対しては，特別な関心を払ってきている．この分野の進歩は，翻訳システムをハードウェア機器と同じように商品として買うことができる点にまで達している．翻訳に関しては，理論的，実用的両方面に対するすぐれた書物が何冊か出版されている［たとえば，Mckeeman ほか［1970］；Gries［1971］†］．本書では，翻訳システムは使用できるかまたは必要な要求を満たすように作ることができることを仮定している．実際，それらは仕様を満足するように作れるハードウェア機器と同じように扱っている．この仮定はある場合には簡単過ぎるかもしれないが，オペレーティング・システムを理解するためには，都合のよい仮定であろう．

計算機システムの持っている2番目に重要な機能は，大量のデータを扱うことのできる能力である．データは**ファイル**（file）と呼ぶ論理的単位として構成し，その使用の方式に合わせて格納する．ファイル内の要素を通常**レコード**（record）と呼ぶが，ファイルを如何に構成するかによって，それらへのアクセスはいろいろの方法が可能となる．たとえば，順次型構成のファイル（たとえば，磁気テープファイル）においては，レコードははじめに格納されたときと同じ順序にアクセスされる．計算機システム内に，長時間，構造化されたデータの大きいブロックを格納しておくことができるという能力は，強力で便利な道具として働く．ファイルは，大抵，主記憶ではなく2次記憶に置く．ファイル・システムがあるために，何人かのユーザが，容易にデータを共有することができる．もし上手に構成してあれば，任意のデータ要素に対する平均アクセス時間を速くすることができる．ファイル・システムは，計算機システムの資源を使用する必要があるので，周辺機器，コア記憶，処理装置の時間などを獲得するために，計算機システムと連絡しなければならない．したがって，ファイル・システムと計算機システムの間のインターフェイスは翻訳システムのところで述べたものと同じように，慎重に設計しなければならない．

計算機システムによってしばしば提供されるソフトウェア機能は，ほかにもいくつかある．数値計算用のサブルーチンのパッケージ，統計処理用サブルーチンのライブラリ，**分類**（sort）・**併合**（merge）用ライブラリなどにより，あるユーザにとっては計算機システムは更に有用なものとなるだろう．しかしな

† 訳注． 訳書：牛島和夫訳・**コンパイラ作成の技法**（日本コンピュータ協会，1976）

がら，一般に応用プログラムを用意しておくことは，言語翻訳システムやファイル処理システムを用意することに比べて，そんなに重要なことではない．そのようなルーチンをつくることは，数値計算法，統計学，データ構造などの問題であり，したがって，本書が扱う領域をはみ出してしまう．

1.2　オペレーティング・システム

オペレーティング・システム(operating system) は，資源の共有を実現するソフトウェアである．全てのオペレーティング・システムが，その操作を，共有を実現するための機能だけに限っているというわけではない．初期の頃は，言語翻訳システムやユーティリティ・プログラムも，オペレーティング・システムの一部であると一般には考えられていた．最近の実現法としては，これらの機能はオペレーティング・システムからはっきりと区別されているけれども，広い見方をする場合も依然として存在する．大抵のオペレーティング・システムは，入出力機器を共有するのに最小限必要なものよりずっと複雑なファイル・システムを用意してある．ファイル・システムは，入出力機器の共有のために必要であるというよりも，むしろ，便利さのために用いられている．すなわち，オペレーティング・システムがそれを持っていると，ユーザはいろいろの入出力機器に対して，簡単にアクセスをすることができる．しかし，ハードウェアの共有に直接責任を持つソフトウェアであるオペレーティング・システムの**核** (nucleus) は，機械が多種多様であってもその機能はかなりのところまでは不変である．核は処理装置，主記憶，周辺機器を共有する手段を用意している．ユーザと接触するようなオペレーティング・システムの部分，たとえば，**バッチ** (batch) **システム**やテキスト編集システムは，ちょうど核が実際の機器をハードウェアの基本要素として使用しているのと同じように，核の機能をソフトウェアの基本要素として使用している．核は主としてオペレーティング・システムの他の要素にいろいろの機能を与え，今度はそれらの要素がユーザにさらに上位の機能を与える．オペレーティング・システムの核を構成し実現する技法を述べることが，本書で扱っている題材の主要部分である．

“オペレーティング・システムのユーザ”という概念は，ややあいまいなものである．非常に簡単にいえば，**ユーザ** (user) とは，システムで処理すべき

プログラムを持っている人間を指す．しかし，そのプログラムは非常に多様な性格をもつ可能性がある．処理装置の時間，記憶空間，周辺機器などへの要求は，資源のほんのわずかの部分から機械全体に対するまで，広い範囲に及ぶ．以後の議論では，ユーザとは処理すべき計算を持っている人を指し，また，その計算は計算機システムの処理能力の範囲に入っているものと仮定しよう．

オペレーティング・システムがどんな機能を提供しようとも，それとは別にある性能の条件を満していなければならない［Hoare および Perrot, 1972］．特に，オペレーティング・システムは次のような条件を満足していなければならない：

1.　信頼性のあること　オペレーティング・システムは，少なくともそれが実行されているハードウェアと同じくらいの信頼性がなければならない．もしも，ソフトウェアかハードウェアに誤りが生じたならば，システムは誤りを検出し，問題点を修正するか，あるいは最悪の場合でもシステムのユーザが誤りによりこうむる損害を最小にする努力をしなければならない．

2.　保護されていること　ユーザは，他のユーザが干渉してくることは望まない．したがって，オペレーティング・システムは，各ユーザがお互の誤りとか故意に試みた違反によって影響を受けないように保護しなければならない．

3.　効率のよいこと　オペレーティング・システムは，一般に自分自身が働くために，ハードウェア資源を大量に使用する複雑なプログラムである．オペレーティング・システムで消費される資源は，ユーザは使用できない．したがって，システムそれ自身が出来るだけ効率よく働かなければならない．さらに，システムはアイドル時間を最小にすること，すなわち**使用率**（utilization）（アイドルでない時間のパーセント）を最大にするように，ユーザの資源を管理すべきである．

4.　予測可能であること　システムに対するユーザの要求は，一般には，予測不可能である．同時に，ユーザは，かなり長期間にわたって，変動の少ないサービスを受けることを望んでいる．すなわち，ユーザのプログラムがシステム内に読み込まれたならば，ユーザは同じプログラムを前回処理した体験から，いつ出力を得ることができるのか大ざっぱな見通しを持たなければならな

い．

5. 便利であること　ユーザは必然的というりも，むしろ，経済的な必要性から計算機システムの共有を行なうのである．ユーザはオペレーティング・システムを絶対に使用しなければならないのであるから，それは柔軟性に富み，便利に使用できるものでなければならない．結局，ユーザが計算機システムに最も密接に結びついている．したがって，システムはユーザのことを念頭において設計しなければならない．

資源の共有を実現し，同時に，上に並べたような性能の目標を達成するために，オペレーティング・システムは計算機システムの資源の割り当てを完全に制御できる状態になければならない．資源の割り当てを独占的に行なうことによってのみ，オペレーティング・システムはユーザの要求と性能の目標との釣り合いを実現することができるのである．

1.3　資源割り当て

オペレーティング・システムは，効率よくかつ信頼性のある操作をするという条件の下で，ユーザの要求とシステムの能力とに応じて，資源の割り当てを行なう．単純なシステムでは，割り当てられる中で鍵となる資源は，主処理装置の時間である．それがなければ，計算は少しも進まない．もっと複雑で強力なシステムにおいては，ハードウェア機器，ソフトウェア・サービス共に，個個に割り当てられる乏しい資源と考えられる．理想的には，ユーザはこれらの資源を必要なときにいつでも獲得することができなければならない．残念ながら，これは常に可能というわけではない．たとえば，ユーザの要求がシステムで使用可能な資源の量を越えてしまうときもある．そのような状況においては，システムは妥協をしなければならず，もっとも重要であると考えられる要求だけを満さねばならない．要求の重要度は，ユーザの要求の緊急度だけでなく，要求を認めたことによって生ずるシステムへの影響によっても決ってくる．ある資源を割り当てると，それを待っている他の要求へのサービスが低下するが，その資源の使用率は向上する．良いサービスを維持しつつ資源を有効に使用することが，オペレーティング・システムの一つの目標である．

唯一つの長所だけで，システムの質を判定することが出来ないというのは，システムが本来持っている性質である．システムは，本来，多次元的なものである．しかし，さらに悪いことには，システムは一般にどの単独の要素が提供するものよりもずっと複雑な機能を提供するものである．このために，システムの分析の仕事が絶望的なほど複雑なものになってしまう．

おそらく，効果のある唯一の考え方は，客観的な評価ができる特定の目標を念頭においてシステムを設計し，手で触れることができないものに関して，最良になることを望むことである．この節では，オペレーティング・システムによって提供される主要な資源の割り当てに関するいくつかの考え方を示す．

1.3.1 処理装置の時間

精巧に設計されていないシステムにおいては，システム全体を単一の資源の単位として割り当てる．ユーザは，機械を支配しているかあるいはそれを支配することを待っているかのどちらかである．この方式を実現するのは簡単であるが，装置を必ずしも効率よく使用しているわけではない．実行中のプログラムは，周辺機器を何も使用しないかもしれない．あきらかに，実行中のプログラムが使用していない資源は空き (idle) のままであり，それらの使用率が低下する．この問題は，ハードウェアにおいて本来備っている**並行性**(concurrency) を利用することにより，幾分軽減される．

周辺装置は主処理装置と並列に操作することが可能であるから，一つのプログラムが入出力操作を実行している間に，他のプログラムを主処理装置で実行することが可能となる．そのような方式を実現する際の詳細は複雑ではあるけれども，この方式を用いれば各機器の使用率をあげることができる．しかし，入出力機器を始動させるためには，プログラムを処理装置で処理しなければならないから，主処理装置の時間の割り当ては難しい問題となる．たとえば，周辺機器を必要としないようなプログラムに処理装置を割り当てることは，これらの機器の空き時間を非常に増大してしまう．しかし，またすべての資源の使用を最適にしようと試みると，ある種のプログラム，とくに，周辺機器を必要としないようなプログラムの処理は，堪えられないほど遅くなる．そのような矛盾はしばしば発生し，しかも，簡単に解決できない．機器の並列性を利用す

るようなシステムにおける処理装置の割り当ての方法については，第3章で検討する．

1.3.2 記憶管理

記憶の管理と処理装置の割り当てとは，密接な関係にある．プログラムは主記憶の中に存在しているときだけ実行可能であるが，処理装置を使用できる機会があまりないようならば，主記憶にとどまっているべきではない．さもないと，それらが使用している記憶領域が無駄になってしまう．主記憶は，高価な資源であり，かつ近い将来もひき続きかなり高価であることが予想される．したがって，システムは記憶を有効に詰めることにかなりの時間を費して，役に立つプログラムをコア内にとどまらせたり，プログラム間の隙間をなくすようにする．

プログラムの間の“空隙(hole)”をなくすために，システムはプログラムを**再配置**（relocate）する技法を使用する．すなわち，プログラムをコア内で移動させ，無駄な空間を減らすのである．記憶の構成を特殊な方法で行なえば，再配置をずっと容易に行なうことができる．このような方式を用いれば，オペレーティング・システムは，コアから情報の出し入れをかなり柔軟に行なうことができる．しかし，そのような方式をとるためには，通常，特殊なハードウェアが必要であり，時にはかなりの量の**オーバーヘッド**（overhead）を生ずる．記憶管理システムが，さらに複雑になってしまってもよいというためには，通常の方式で行なうのに比べて，記憶の使用効率が大幅に改善されなければならない．残念なことに，複雑な記憶の割り当て方式が，全体のシステムに対してどの様な効果があるのかということを定量的に述べるのは困難である．記憶の割り当て方式に対する研究は非常にたくさんあるけれども，最適な方法を解析し実現することは未だ難しい問題である．

記憶の割り当てに関する技法は，第4章で紹介し，第5章でそれらの解析を行なう．

1.3.3 入出力機器

入出力機器およびそれに付随する制御装置と，チャネルとを割り当てる方法

は，機器の速度によって大きく左右される．カード・リーダ，ライン・プリンタ，磁気テープ駆動装置などのような機器は，一時に，一つのジョブに割り当てられる．それらのジョブは，それが必要な間ずっと周辺機器を使用し，それが終ってから使用を放棄する．磁気ドラムやディスクのような高速の**乱アクセス機器**（random access device）は，入出力操作を単位として共有することができる．すなわち，装置は長い期間一つのジョブに割り当てるかわりに，複数個のジョブが並行的に機器を使用することを許す．2つのジョブが，一つの機器に対して同時に入出力操作を行なおうとすると，遅れが生ずる．しかし，これらの装置は高速であるから，その遅れは気にならない．カード・リーダや磁気テープ駆動装置などは，一つのジョブから他のジョブへ切り替えるときに媒体の装填をやり直すことが物理的に必要になってくるので，後者のモードで操作することは不可能である．

機器の割り当ての方法は，システムの正しさと効率とに，重大な影響を与える．たとえば，1台の読取り機と1台の印刷機とから成るシステムにおいて，不適当な割り当て方式を用いると，予想していなかったような停止状態に陥入る可能性がある．ユーザAが読取り機を保持したまま，自分が終了するために印刷機を必要としているとき，ユーザBが印刷機を保持したまま，自分が終了するために読取り機を必要とするならば，AとBとは永久に（あるいは，どちらかのユーザが他方を終了させるために，自分の保持している資源を解放するまで）待ち続けることになる．資源割り当てシステムは，そのような状況が生ずることを防がなければならない．

割り当て方式は，機器の使用効率にも影響を及ぼす．たとえば，3つのテープ駆動装置から成るシステムにおいて，ジョブAは2台，ジョブBは3台の駆動装置を要求する場合を考えてみよう．2台の駆動装置が空いていて，1台は未だ使用中のとき，Aに直ちに2台の駆動装置を与えることよりも，Bの要求を満すように3台目の駆動装置が空くのを待っている方がよいかもしれない．もしも，Bが高価な資源（他の周辺装置か，大容量のコア記憶など）を既に保持しているならば，2台の駆動装置が一時的に空き状態になっているけれども，Bをシステムから早く追い出した方がずっと効率的であろう．このようなスケジューリングの決定は，局所的な見地からみれば直観に反するけれども，実際

には全体的な効率を上げることになる．

効率的な周辺装置の割り当ては，次の2つの理由で実現することが困難である．第一に，幾つかの異った型の機器を最適に割り当てる方法を見つけ出すような解析は，現在の数学的技法では不可能だからである．特殊な方法は多く知っているけれども，多種類の資源システムのバランスをとるのは，未だ，科学的というよりも名人芸的なものである．第二に，与えられたシステムにおいて，資源割り当て方式の性能を測定することが困難だからである．性能が“悪い”ということを知るだけでは，必要な問題点を捉えることはできない．残念ながら，現在使用可能なモニタの技法では，システムの特定の非能率な点を見つけ出すことは不可能であり，適切でない割り当てであることを一般的に示すだけである．主記憶と中央処理装置との相互作用が関係している場合は，解析的技法，経験的技法ともに期待することができない．処理装置や記憶を他の資源と分離して考えるときには，効果的な方法があるけれども，これらの割り当て方法が複雑なシステムの中で混在しているときは，その性能を予測することは困難である．

入出力機器，制御装置，チャネルを共有する方法は，ほとんど機器に依存して決まると言ってよい．一つの可能な方式を第6章で議論する．

1.3.4　ソフトウェア資源

よく使われている広汎なオペレーティング・システムは，よく整備された応用プログラムのシステムおよびユーザ・ライブラリを用意していることが多い．割り当てるべき資源として，これらのライブラリはハードウェアと多くの共通点を持っている．コンパイラ，インタープリータ，テキスト・エディタや，分類併合システムなどの特殊なユーティリティは，リエントラント型に作られていれば共有することができる．プログラムが，データと手続きとの部分に完全に分離して作られているとき，**リエントラン**（reentrant）であるという．すなわち，“格納”操作（“store” operation）は，データの部分にのみ行なわれる．データ部分の写しを各ユーザ個人用に作り，手続き部分の写し一つだけを皆で共有することにより，エリントラントのコードを多数のユーザで共有することができる．プログラムがリエントラントになっていなければ，ハードウェアの

機器と同じように，それぞれの写しは，一時には，一人のユーザにだけ割り当てなければならない．ソフトウェアの共有は，必要な時に，妥当な代価を支払えば余分の写しを作ることができるという意味で，ハードウェアの共有に比べて，ずっと柔軟性に富んでいる．ハードウェアに対しては，一般に，存在するものだけを扱うようにしなければならない．ソフトウェアの共有においても，ハードウェアの場合と同様に，効率と正当性の問題を考慮しなければならない．

1.4 スーパーバイザ

オペレーティング・システムの設計においては，重要な機能を生み出す幾つかのプロセスを**スーパーバイザ**（supervisor）と呼ぶ“支配者”プログラム（“master” program）の下に集めるのが伝統的な方法である．中央集権者として，スーパーバイザはシステムの残りの部分の結合を実現する．それは特権を認めたり，時には，罰を与えたりしながら，協力して働くプログラムを組織化する．すなわち，プロセスや実際の機器との間の通信や同期の手段を提供する．従来から，プロセス間を渡されるメッセージ，機器の起動や停止，ハードウェアの信号などは，全てスーパーバイザで扱かっている．（あるシステムでは，その時間の半分位をスーパーバイザのために費している．）しかしながら，スーパーバイザが非常に大きくなってよいという本質的な理由は，何もない．分散型の構成の方が，システムの流れ作業的操作に対しては有利となるであろう．

スーパーバイザは，一般に，システムのすべての資源および機能を，ユーザ全体に割り当てる管理をする．言語翻訳や，ファイル・システムの保守などのようなある種のシステムの機能が大きくなり過ぎると，それはスーパーバイザから分離する．形態としては，スーパーバイザの“外側”にあるものであるが，それはスーパーバイザが要求することを行なう責任をもっている．大抵のシステムでは，プログラムは，たとえ，スーパーバイザに直接関係した部分ではなくても，スーパーバイザにシステム内の資源を使用することを知らせなければならない．したがって，たとえスーパーバイザが小さいものであっても，それはシステムの管理をほとんどつかさどっているのである．

1.4.1 スーパーバイザの仕事

オペレーティング・システムの主たる目的は，共有を実現することである．これを効率よく行なうために，スーパーバイザはオペレーティング・システムが提供する機能を管理している．スーパーバイザの仕事は，大よそ次の4つの分野に分けられる．すなわち，制御と監視，相互通信，保護と強制，ユーティリティである．[Mealy, 1967].

1.4.1.1　制御と監視

スーパーバイザはシステム内のジョブを管理し，順序づけをしなければならない．ジョブの実行のスケジューリング，ジョブが使用した資源の料金計算，ジョブ制御言語の解釈などは，すべてジョブの処理に対応する．これらの仕事はスーパーバイザとは独立に実現することも可能であり，あるいはまた，スーパーバイザの下に包含することも可能である．ユーザは，自分の要求をジョブ制御言語を用いて指定しておく．スーパーバイザは，その要求を解釈しいろいろの資源割り当てシステムに対し，ジョブがどんな資源や仕事を要求しているのかを伝える．資源割り当てシステムは，それらの要求を優先度に従って，満してゆく．料金計算の手続きは，ユーザによって使用された資源の量を記録し，のちほど適当なユーザのコードに対して料金を請求する．

1.4.1.2　相　互　通　信

スーパーバイザは，異ったプログラム同志がつながるために，接続機構 (link) を提供する．2つの独立したプログラム，たとえば，ファイル・システムと記憶管理システムが相互通信を望んだとき，それらのプログラムはスーパーバイザに接続してくれることを要求する．スーパーバイザが最初の接続を確立した後は，プログラムはシステムで定義した機構を用い，たとえば，共通の“郵便受け (mailbox)”を通して“文字”をやり取りするようにして，相互通信を行なうことができる．相互通信の際は，スーパーバイザを介して初期設定を行なわなければならない．何故ならば，個々のプログラムがどの位置に存在し，どのようにすれば接続できるかという特別の情報を知っているのは，スーパーバイザだけだからである．スーパーバイザはそのような特別の情報を持っているために，他の相互通信の仕事，たとえば，システム・プログラムとユーザ・プログラムとの接続なども実行している．スーパーバイザの能力の大部分はそのユーザに比べて，多くの情報を持っているということだけから生れてくるのは，

確かである.

1.4.1.3 保護と強制

処理が順調に行なわれることを保証するために，スーパーバイザはシステムおよびユーザの両者に対し，制限条件を遵守させなければならない．あるルーチンの集団は，ハードウェアの信号，特に，異常条件の信号を監視している．誤りの条件が発生すると，通常，割り込みを介して，スーパーバイザはジョブに制約を加えたり，ジョブを消滅させたりする．実行時間と出力量の制限は，厳守されなければならない．スーパーバイザは，また，特別な“切符”を発行して，保護機構を実現する．保護されている資源は，おそらく偽造はできないような正しい切符を提示することにより，はじめて使用することができる．

情報を大量に集中していることから考えると，スーパーバイザはシステム内で非常に重要な部分を構成していると言える。したがって，スーパーバイザは，システムの他の部分やユーザに対して注意を払って，その記憶域が偶然にしろ故意にしろ，めちゃめちゃにこわされることを防止しなければならない．スーパーバイザの保護を行なうために，あるシステムでは2つの実行モード，すなわち，スーパーバイザ・モードとユーザ・モードを設けている．入出力操作の起動とか記憶の特別な領域を使用するような特権命令は，スーパーバイザ・モードでしか実行できない．この機構により，スーパーバイザは非特権的システムやユーザ・プログラムに対して，特別な能力を持つことができる．重要な情報は，スーパーバイザ・モードでしか使用できないような保護された記憶に格納することにより，スーパーバイザは自分を保護できる．このようなハードウェアの機構は，スーパーバイザが破壊されることを保護しているのである．

1.4.1.4 ユーティリティ

資源を割り当てる以外に，スーパーバイザは幾つかの機能を用意している．それは，緊急用ダンプ，プログラム・ライブラリの使用，オン・ラインのメッセージの処理などである．相互通信用の記憶領域や保護システム用のテーブルなどは，スーパーバイザの内部にあることが多い．また，ジョブ制御言語の処理，システム・コンソールのインターフェイス，料金計算などもある．もっと細かい機能が必要になったときは，スーパーバイザは，一般に特殊なソフトウェアのプロセサを呼び出す．

1.4.2　集中型制御方式の評価

システムの重要な機能をスーパーバイザに集中してしまうという方針には，利点と欠点とがある．集中化の一つの利点は，保護機能を簡単に実現できることである。スーパーバイザ・モードを用いれば，システムのある部分を，ユーザから隔離することができる．もう一つの利点は，構成が単純化するということである．重要なシステムの機能をシステム内に散在させないで，スーパーバイザの下に集めることは，考え方としては容易である．この方針は，システムが非常に小さく作られているときにうまく働く．何故ならば，システムの制御機構を，簡単に位置を決めることのできる境界内に集中できるからである．残念ながら，この方式では多くの機能を作りたくなるが，その場合にはスーパーバイザとオペレーティング・システムとの間の区別をしたという利点が失われてしまう．

しかしながら，集中型のスーパーバイザには，重大な欠点がある．プログラムは自分自身で相互通信することが許されていない．したがって，何か重要なことが生ずるたびごとに，スーパーバイザを呼び出す．同時に，多数のプログラムが相互通信をしようとすれば，そのうちの少数のものだけがスーパーバイザに連絡できるだけである．これは，**隘路**（bottleneck）の一つの例である．すなわち，非常に大量の情報が制限された容量しかない通路を通らなければならない．スーパーバイザを呼び出しそのサービスを受けることは，無駄を作り出す．スーパーバイザは頻繁に呼び出されるので，それはシステム内の主要な（あるいは，唯一つの主要な）隘路となり得る．

効率の面から考えるほかに，多くの機能をスーパーバイザにまとめてしまうと変更が困難になる．機能は相互に依存し合っており，ある機能を分離することは困難である．スーパーバイザの一部を変更すると，それに関係したところすべてに影響が及ぶ．このようにして，簡単な修正でも全体の分解修理を引き起してしまうのである．

これらの問題にもかかわらず，最近の研究の方向は，スーパーバイザを用いるかあるいは協力して働くスーパーバイザ [Brinch Hansen, 1970] の集団を用いることを，はっきりと支持している．たとえば，プログラムは主記憶，ディ

スク，カード・リーダ，ライン・プリンタ，端末などの主要なハードウェアの資源を付与することができる．相互に協力し合いながら，これらのプログラムが一緒になって，スーパーバイザと同じ機能を実行する．単一のプログラム内に機能をモジュール化することにより，各構成要素の複雑さを減少できる．しかし，一方で，各構成要素の相互通信の量を増大させ成功しているシステムでは，集中型および協力して働くスーパーバイザの両方を用いて設計してきた．システムの構成のいろいろな方法については，第2章，第8章および第9章で述べる．

1.5 お わ り に

計算機システムにおけるオペレーティング・システムの機能は，一つの会社における経営管理の機能に比較し得る．それは，引き起される無駄と引き換えに，重要な機能を作り出す．管理をする人は，会社が行なう有益な仕事を組織し，働く人々の集団に，責任をもって会社資源を分配する．企業におけるオートメーションの時代が到来したのにもかかわらず，管理に対する必要性は依然として残っている．同じことは，計算機システムに対してもいえる．ハードウェアが，ずっと複雑な仕事をすることができるようになっても，オペレーティング・システムに対する必要性はなくならない．しかし，オペレーティング・システムが行なっている仕事の詳細な部分は，かなり変化していくだろう．

ある種の資源，特に，記憶や周辺機器の管理をする低レベルの技術の多くのものは，そう遠くない将来にはハードウェアによって行なわれるであろうことが期待されている．しかし，これは提示される問題がなくなってしまうということを言っているのではない．むしろ，資源の共有の問題は，常にわれわれの目の前からはなれない．資源の詳細な仕様は変更するかもしれないけれども，それらを共有する基本的な技術はこれからの長い間にわたって，計算機システムの設計の多くの分野において，広範囲の応用性を発揮するであろう．

演 習 問 題

1.1 記憶装置の特性に関して，速度，1ビット当りの価格についての一覧表を作製せよ．この表を，1950年，1955年，1960年，1965年，1970年の各年

代について完成せよ．また，表から読み取れる傾向を議論せよ．

1.2 多くの入出力機器が，計算機に接続できる．異った種類の入出力機器の一覧表を作り，速度と1字当りの価格に関する特性を示せ．特殊な装置，たとえば，COM(Computer Output Microfilm)，文字読み取り機，マーク読取り機などと，カード・リーダ，ライン・プリンタなど伝統的な装置とを比較せよ．

1.3 最先端（front-end）のデータ収集装置あるいはネットワークのインターフェイスなど，特殊装置として働いているミニコンピュータ(minicomputer）を考えよ．そのような計算機システムのオペレーティング・システムでは，どのような機能が必要であるか．汎用システムで用意されている機能のうち，どんなものがその重要性を失うか．

1.4 チャネルと，新しいミニコンピュータとの間の差異は何か．ミニコンピュータはチャネルとして働くことができるか．それは中央計算機に対して，どのように接続されるであろうか．

1.5 ある特定の施設で用いられた装置の一覧表を，購入価格をつけて完成せよ．処理装置，記憶装置，周辺装置の価格を比較せよ．各装置の価格と計算センター全体の予算とをどのように比較したらよいか．

1.6 自分の近くにある計算機施設において，オペレータが働いているところをしばらく観察せよ．オペレータに対するジョブの申込書を用意せよ．自分がシステム設計者ならば，どの程度，オペレータを信頼するか．システムがすべき決定の例をあげよ．どんな仕事をオペレータに与えれば，一番良いか．

1.7 上方向に互換性を持ち，規模と速度が増大してゆくような一連の機械に対するオペレーティング・システムの設計を依頼されたと仮定せよ．全体で一つのオペレーティング・システムを設計するのか，あるいは，各機械毎に一つづつのオペレーティング・システムを設計するのか．市場性，保守，設計，実現などの観点から両者の得失を論ぜよ．

1.8 ある計算機システムが，少量の入出力の要求を持つ数値解析用に用いられていると仮定せよ．このような環境が，計算機システムの**アーキテクチャ**（architecture）およびオペレーティング・システムの機能にどのような影響を与えるか．

1.9 新しいミニコンピュータに対するオペレーティング・システムの設計を依頼されたと仮定せよ．システムに用意したいと思う機能の一覧表を作製せよ．重要度の高い順に，機能を並べよ．

1.10 Grosch の法則によれば，計算機システムの能率は投資価格の二乗に比例するといわれている［Sharpe, 1969］．これは，大きな集中型の計算機の設備を作ることを正当化する主要な拠りどころとなっている．Grosch の法則が，CPU，計算機システム，ミニコンピュータ，通信装置などに関して成立していることを確かめてみよ．

1.11 経済における需要と供給の関係は，価格機構によって制御される．計算機の機能に関してのいろいろな価格機構の相対的利点を議論せよ．公正，平等で，能率くよ実現できるような価格機構の概要を述べよ．

第 2 章

プロセス

2.1 はじめに

本章の主題は，オペレーティング・システムにおける計算の基本的な単位である**プロセス**（process）を定義し，その性質を述べることである．最初に，よく知っている型の計算，すなわち，**逐次型プログラム**（sequential program）を考えよう．ALGOL や PL/I で書かれた大きな逐次型プログラムは，**手続き**（procedure）と呼ぶもっと小さな逐次型プログラムに分割できる．帰納手続きが許されていないならば，一つの手続きは，一時には一回しか生かされない．したがって，実行中のどんな計算も，現在生きている手続きにより一意的に識別することができる．しかしながら，オペレーティング・システムにおいては，この様な計算と手続きの間の単純な関係は失われている．一つの手続きは，同時に，システム内のいくつかの場所で実行されているかもしれない．たとえば，いくつかのディスクのチャネルは一つの入力手続きを共有しているかもしれない．また，逆に“入力”というような概念的には一つの操作であるような機能が，それが実行されるごとに複数個の異った手続きを使用するかもしれない．ジョブが，“カードを読め”と命令しても，次のカードがどこに在るかによって，システムは異った動作をする．それはすでにコア記憶にあるかもしれない．ディスク内にあるかもしれない．あるいは，また，カード・リーダから物理的に読み込まなければならないかもしれない．ある場合には，“カードを読め”という命令は並行的に実行されるステップを含んだ多ステップの操作かもしれない．したがって，与えられた計算の仕事に対して関係している手続きは，一つ以上存在するかもしれない．あるいはまた，一つの手続きが一つ以上の計算

の仕事において，生かされているかもしれない．逐次型プログラムに見られる手続きと計算との一対一の対応関係は，オペレーティング・システムにおいては存在しない．したがって，計算と手続きとの概念は等価なものとはなり得ない．

オペレーティング・システムにおける計算は，本質的に逐次型であるというわけではない．中央処理装置，チャネル，周辺機器はすべて並行的に動いている．ソフトウェアがこの並行性を考慮に入れているのは当然である．典型的な場合，オペレーティング・システムは入力，出力，CPU の動作など，ほとんど独立に処理される幾つかの計算から成り立っている．このような“独立した計算”の概念を形式的に**プロセス**というのである．プロセスはオペレーティング・システムの内部で何が起っているのかを理解するために便利なモデルである．計算機システム自身に組み込まれている並行性を調べれば，並行処理プロセスを用いたくなる動機が最もよく説明がつくであろう．

2.1.1 ハードウェア・システムにおける並行性

ハードウェアの観点からみると，大抵の場合，ビットのレベルにおける並列処理が使われてきた．従来から，加算器や乗算器などの演算装置の速度を上げる方法として，並列に処理できるような部分をふやすことを用いている．このような方向をとる理由は，充分にある．ディジタル回路の現在の技術では，個々の素子の速度は，その物理的限界に近づきつつある．回路は10ナノ秒のオーダーで動作するが，光は１ナノ秒で約 30cm 進む．さらに小型化を進めれば，回路の速度を幾分高めることになるであろう．しかしながら，機械の計算能力を大幅に改善するためには，恐らく，並行性の利用をすることになるであろう．

周辺機器およびチャネルは，中央処理装置と並行的に動いている．中央処理装置は，またハードウェアの並行性を利用している．多くの機械は処理装置の数をふやすことによって，その処理能力を増大させた（たとえば，ILLIAC Ⅳ など）．複数個の演算装置と多数のチャネルとが，すべて同時に働くようなハードウェア・システムにおいて，そのような装置を用いた計算を組み立てることは複雑な仕事である．それを効率よく組み立てることは，さらに，困難である．

一人のユーザは，通常，システムの機能を一様に利用するわけではない．たくさんの入出力を行なっているならば，恐らく，演算装置はそうたくさん使うことはないであろう．また逆に多くの計算をするのであれば，入出力装置は，恐らく空いたままになっているであろう．同時に複数個のユーザを生かしておくことにより，システムがその装置を充分に用いる可能性が高まるのである．一人のユーザが印刷をし，次のユーザが読み込みをしているときに，三人目は中央処理装置で計算をしている．したがって，並行的に動くハードウェアの装置を用意するとともに，並行的に実行できるユーザのジョブがいくつかなければならない．

2.1.2　ソフトウェア・システムにおける並行性

ユーザ・ジョブは並行的に処理されているとき，一般にはお互に相手の存在を意識してはならない．ジョブが印刷の命令を出したとき，他のジョブが現在カードを読み込み中であるということには無関心である．ハードウェアの装置も同じような動き方をする．印刷機は，通常，読取り機の存在には関心を払わない．このような状況を正しくモデル化するために，このような並行性を考慮に入れた計算の単位が必要となるが，それがプロセスである．計算は独立に進行するのであるから，計算をあらわすプロセスも独立でなければならない．さらに，装置はそれぞれ異った速度で動くから，プロセスの相対的な速度も独立でなければならない．したがって，計算機システムを考えることにより，プロセスの性質が幾つか摘出できる．すなわち，プロセスとはいろいろの速度で並行的に処理されるような独立な計算である．

計算は，ときどき，情報を交換しなければならない．計算はお互に通信し合わなければならないという意味では，完全に独立であるというわけではない．計算をモデル化するのに用いるプロセスは，このような通信の能力を考慮に入れてなければならない．もしもでたらめに通信し合うことをプロセスに許すならば，それらの複雑な相互作用を理解することは困難である．したがって，陽にきちんと定義された機構を用いたときだけ，プロセスは相互通信を行なえるものとする．プロセス間の相互通信の方法は，後節で述べる．

ハードウェアとソフトウェアの構成要素の間には並行性があることが示され

たが，プログラムは，通常，逐次型である．これは常に正しいとは限らない．ある種のプログラミング言語においては，並行処理を指定することを許している．しかし，人間の頭脳では，逐次型プログラムを習得する方がずっと適しているように見える．したがって，本書ではプロセスは逐次型であると仮定した．すなわち，プロセス内の操作はきちんと定義された順序に従って適用される．直観的には，逐次型のプロセスとは ALGOL プログラムの実行と考えてもらえばよい．以後の節では，“プロセス”という言葉は，いちいち断わらなくても“逐次型プロセス”の省略形として用いる．

2.2 プロセスの定義

プロセスの概念は，多くの文献にみられる．恐らくオペレーティング・システムにおいて，最も頻繁に使われており，また，最もいい加減に定義されている術語であろう．大ざっぱに言えば，**プロセス**とはある規則に従って内容が変る記憶のセル (cell) の集合と考えられる．計算機においては，このような規則は**プログラム**によって指定され，**処理装置**によって解釈される．底に流れる意図は明確なものではあるけれども，その定義はかなり変ってきている．ほとんどの新しいシステムやそのモデルにおいて，少しづつ違った意味を持つ定義がなされてきた．OS/360 では，“タスク (task)”という術語が用いられ，“制御プログラムの下での多重プログラミングの基本単位”と定義されている [IBM, 1965]．Saltzer [1966] は，“プロセスとは，擬似処理装置 (pseudo-processor) によって実行しているプログラムである”と述べている．Dijkstra [1968b] は，逐次型プロセスとは計算機において逐次型プログラムを実行しているときに生ずるものと定義している．HorningとRandell [1973] は，プロセスの概念に対する形式的な定義を作った．この概念を明確にするために，Horning と Randell [1973] に非常に似た，プロセスの形式的モデルを提示する．

$X=\{x0, x1, \cdots, xN, \cdots\}$ を状態変数の集合（無限集合でもよい）とし**状態変数集合** (state variable set) と呼ぶ．**状態** (state) とは，状態変数集合の各要素に値を割り当てたものと定義する．状態変数の集合に対する**状態空間** (state space) は，その変数の集合が取り得る状態の集合から成り立つ．**作用** (action) は，変数の集合に属するいくつかの変数に対して，値を割り当てる．

状態空間から作った状態の系列を**計算**（computation）という．計算を作り出す一つの方法は，いろいろの作用を連続的に適用することである．各作用は新しい状態を作り出し，状態の系列が計算となるのである．**作用関数**（action function）とは，状態から作用への写像を与える関数である．作用関数も初期状態を与えれば，計算を生成する．それは各状態において適用すべき作用を指定するだけであり，その作用は新しい状態を作り出す．作用関数は，無限に新しい状態を解釈するので，状態の列が生成されるのである．直観的には，変数の集合は記憶に，状態は記憶の内容に，また，作用関数はプログラムに対応する．

ここで，**プロセス**とは，状態空間，その状態空間における作用関数，**初期状態**（initial state）と呼ぶ状態空間内のいくつかの特殊な要素とから定義できる．二変数xとyとを持ったプロセスPを考えよう．Pの計算は，状態の系列によって記述できる．たとえば，$\{(2,1), (4,2), (6,3), (8,4), \cdots, (2i,i), \cdots\}$などである．$P$の計算は，また，状態空間 $\{(i,j)$ ただし，iとjとは自然数$\}$，初期状態（2，1），すべての状態に対して作用 $(x,y)\rightarrow(x+2,y+1)$ を生成する作用関数とで記述することができる．

プロセスの各状態は，そのプロセスが実行している計算の過程における“スナップ・ショット（snapshot)”を表わしている．プログラムとそれが用いる変数が，主記憶内に格納されていると仮定しよう．プログラムの実行は，記憶と，プログラムに付随しているレジスタとを観測することによって視覚的にとらえることができる．任意の時間において，各記憶場所はある値を持っている．プログラムの実行の状態はある瞬間におけるそのすべての記憶場所の値によって与えられる．これが，状態の概念である．状態の系列は与えられた環境におけるプログラムの計算の進行過程をあらわしている．プログラムすなわち作用関数は，その初期状態と共に状態の系列をあらわす別の方法を与えている．

プロセスのある状態については，それを観察する者にとって関心のないことがあるだろう．たとえば，手続きが呼び出されたとき，それを呼び出したプログラムから見て興味のある状態は，その呼び出し時と戻り時，特に，パラメータについての情報である．手続きの“内部的”状態については，特別な注意を

払うことはほとんどないであろう．何故ならば，呼び出しているプログラムの最大の関心事は手続き呼び出しの結果である．同様にして，変数のうちのあるものにだけしか特別の関心を持っていない．すなわち，計算の結果として戻される変数についてだけである．このような考え方は，状態や状態変数の部分集合だけを考えることにより，簡単に形式化される．

前述したモデルは，前節で述べたプロセスの性質をよくあらわしている．たとえば，プロセスはほとんど独立であると前に述べた．明らかに，2つのプロセスの状態変数に共通部分がなければ，それらは相互作用をしない．プロセス間の通信は，共有変数を用いることによりモデル化できる．複数個のプロセスが，同じプログラムのコードを実行するかもしれないと注意しておいた．これは，等価な作用関数が異った変数の集合に作用していることに相当する．確かに，プロセスに関する大抵の概念は Horning と Randell のモデルを用いて述べることができる．われわれの目的にとっては，形式的な定義が使用できることこそ重要な点なのである．本章の残りの部分は少し形式的ではないけれども，それは考察の中に精密さが欠けていてよいという理由にはならない．むしろ，精密な概念が，主として説明が便利になるように，非形式的に扱われているのである．

プロセスは抽象的なものとして概念化してきたけれども，それにより，オペレーティング・システムの実際の構成要素をモデル化することができる．作用関数と状態変数は，手続きと記憶場所をあらわしている．プロセスをソフトウェアの構成要素として実現する一方法を，次節で述べる．

2.3 プロセスの実現

次に述べるプロセスの概念の仮説的な実現法は，ある特定のシステムのものをそっくり写したものではないけれども，多くのシステム，たとえばＲＣ4000，THE, SUE プロジェクト，Venus（第10章参照）などで実現されたプロセスと非常に似たものである．実際のシステムにおいては，プロセスはハードウェアとしての物理的な処理装置の上で，一つまたはそれ以上の手続きを実行することをモデル化している．ソフトウェアの各プロセスは，**プロセス・ディスクリプタ**（process descriptor）と呼ぶデータ構造を用いて一意的に識別する．代表

的なシステムにおいては，プロセス・ディスクリプタには次のような内容が書かれている：

1. プロセスの状態を定義する状態変数（たとえば，実行可能，実行中，停止など）.
2. 退避用領域．ここには，プロセスが実行中に割り込みが生ずると，その時点のレジスタの値が置かれる.
3. プロセスが所持しているか，または，使用することを宣言している資源に関する情報.

ディスクリプタは，また，他のプロセスと相互通信をするために用いる領域も含んでいる．相互通信に関しては，後節で議論する.

第2.2節では，プロセスとは計算の抽象的単位であり，オペレーティング・システムにおける作業を記述するのに便利なものであると述べた．プロセスの機械における表現は，ディスクリプタと記憶の1ブロックとから構成される．抽象的な意味のプロセスと，ソフトウェアとしてのプロセスとを区別することは重要である．ソフトウェアのプロセスは，しばしば，プロセスと言われるし，またその逆も行なわれる．ただし，常に，そうであるとは限らない．この節の残りの部分で，“抽象的プロセス”の姿を明確にしてみよう．本書の議論は，ほとんどすべての場合，ソフトウェアのプロセスを中心にして行なわれるので，“プロセス”という術語は，ソフトウェアのプロセスだけを指すことにしよう.

ところで，ソフトウェアのプロセスを構成するディスクリプタと記憶とは，機械で使用できる資源から割り当てなければならない．ソフトウェアのプロセスを生成するには，2つの方法がある．システムは，常に存在しているある固定した数のソフトウェアのプロセスを持っているように作ってあるかもしれない．したがって，ソフトウェアのプロセスはシステムの生成と同時に作られている．計算を実行するためには，存在しているソフトウェアのプロセスのうちの一つの制御権を得るだけでよい．ソフトウェアのプロセスの数以上に，抽象プロセスが存在する．抽象プロセスは計算を実行するためには，ソフトウェアのプロセスが使用可能になるまで待たなければならない．別の方法として，必要なときにソフトウェアのプロセスを生成したり消滅したりする機構をシステ

ム内に組み込むことができる．この機構は，それ自身ソフトウェアのプロセスではないけれども，これによりシステムはソフトウェアのプロセスを処理することができる．この機構のことを**核心**（kernel）という．この節の残りの部分では，核心が存在しシステムはすでに一個ないしはそれ以上のソフトウェアのプロセスから成り立っているという観点に立って議論を進めてゆく．

プロセスは他のプロセスの生成または消滅をすることができる．一つのプロセスは核心に要求して，別のプロセスを生成することができる．核心は新しいプロセス・ディスクリプタを初期設定し，プロセスに対してある記憶を割り当て，生成を要求したプロセスに対しその作業の終了を知らせる．生成を要求したプロセスは，プログラム（すなわち，作用関数の表現）を新しいプロセスの記憶に格納する．核心は**開始**（start）命令を用意しているが，これはプロセスに処理装置を割り当てることを指示する．生成したプロセスをパラメータとして開始処理を呼び出すことにより，プロセスの生成者は生成したプロセスの実行を開始することができる．同様にして，**停止**（stop）命令は処理装置の割り当てを解除し，**消滅**（destroy）命令はプロセスのディスクリプタと他の資源との割り当てを解除する．

単一の処理装置しかないようなシステムにおいても，複数個のプロセスが存在することがある．核心は各プロセスに対し，適当な時期に処理装置の使用を許す．たとえば，10個のプロセスがあるとき，核心は各プロセスに対して，毎秒の10分の1づつの処理時間を与えればよい．各プロセスは自分自身のCPU（たとえ遅いものであっても）を持っているように振る舞う．実際に，すべてのプロセスが，一つのCPUを共有するのである．プロセスの開始と停止は，共有している処理装置へのアクセスを認めたり取り消したりすることにより実現される．

プロセスは相互に関係のない単位として存在するかもしれないし，また，構造を持つように関係づけられているかもしれない．構造を持たないシステムにおいては，システム内の一つの部分（通常は，スーパーバイザ）が，必要なときにプロセスを生成する責任を持つ．一つの仕事が生ずると，スーパーバイザはそれを実行するためにプロセスを生成する．その仕事が完了すると，そのプロセスは消滅する．したがって，すべてのプロセスはシステム内の構造として

は，“平等”なわけである．すべてのプロセスはスーパーバイザが生成し，消滅させる．もっと構造的に作られたシステムにおいては，すべてのプロセスは平等ではない．たとえば，プロセスは木構造を持つように構成することができる [Brinch Hansen, 1970；Sevcik ほか，1972]．

木構造を持ったシステムにおいては，プロセスはそれが生成したすべてのプロセスの**親** (father) と呼び，またそれを生成したプロセスの**子** (son) と呼ぶ．**祖先** (ancestor) と**子孫** (descendant) も，普通の意味に定義される（すなわち，祖父，孫など）．システムの**生成木** (creation tree) は，次のような有向グラフで定義される．すなわち，**図 2-1** に示すように，各プロセスは**節点** (node) であらわし，AがBの親であれば節点Aから節点Bへの**弧** (arc) でつなぐ．生成木は，システム内の任意の時点におけるプロセスの順序関係を記述している．

木構造を持ったシステムは，資源の受け渡しと制御権の位置に関して，非常に厳密な規則を持っている．たとえば，各プロセスが生成されたとき，その親が“所有”している資源だけしか受け取ることができない．さらに，親はその子の動作に関しての制御権を持っており，異常状態を修正する手段をとることができる．特に，親に対してはその子のプロセス・ディスクリプタを調べて変更する事を許し，その状態を変えることができるようになっている．

このような高度な構造方式には，明らかに，幾つかの利点が存在する．第一に，資源の割り当てが厳密に制御できる．各プロセスの資源は，ある時点では，

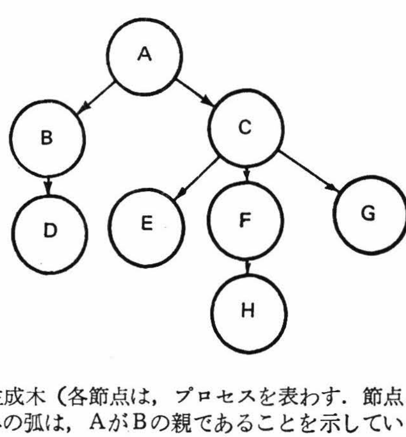


図 2-1 生成木（各節点は，プロセスを表わす．節点Aから節点Bへの弧は，AがBの親であることを示している）

その祖先の所有物であった．もしも，プロセスがその資源を幾つか解放したいときには，それらを親にだけ返却することができる．どのプロセスがどの資源を所有しているのかという点に関するあいまいさは，決して生じない．第二に，全体の構造により，あるプロセスは他のプロセスよりも強い制御権を持ち，仕事の分割に関する簡単な機構を作り出している．プロセスがある仕事を割り当てられたとき，子のプロセスを生成しそれらに仕事のある部分をまかせることができる．これらの子のプロセスは仕事が完了するまで，まかせられた仕事をさらに分割するかもしれない．また，前にも述べたけれども，この構造はどのプロセスが制御権を持つのかということ，すなわち，親が子の制御権を持っていることを常に明確にしてくれる．構造のないシステムにおいては，スーパーバイザが何でも知っている．すなわち，すべてのプロセスは同一の親を持つのである．集中管理型では，仕事の分割はもっと自由に行なえるけれども，スーパーバイザはシステム内のすべてのプロセスの追跡をしなければならないので，情報の整理の仕事が複雑になる．大抵のシステムは，ある程度まで構造化されている．あるプロセスの集団の間での局所的な約束ごとにより，あるプロセスは，それ以外のプロセスとは違った能力を持つことができる．次のレベルがその生成者よりも能力が少ないようなレベル構造を持つシステムもある．これは，大ざっぱに言えば，長い鎖状に縮退している木構造に対応していると考えられる．構造を作り出すこれらの技法には，利点と欠点とがある．木構造とレベル構造の例は第８章と第10章とで述べている．

これまでの話では，プロセスは相互通信する手段を持っていなかった．一度生成され開始されたプロセスは，停止し消滅するまでお互に独立に処理をしている．しかし，プロセスは情報を交換する必要性があるであろう．情報の交換を行なわせるために，プロセス間の相互通信の機構を用意しなければならない．

2.4 プロセスの相互通信

ハードウェアとソフトウェアの資源はそれを効率よく使うために，共有しなければならない．単一のソフトウェアのプロセスとして，システム内で実行しているユーザのジョブを考えよう．そのようなプロセスが存在している間，あるハードウェアの機器をずっと専有していなければならないような状況は稀に

しかない．そのプロセスが周辺装置を使用するために，チャネルを用いていると仮定しよう．一般に，チャネルが入出力要求を処理する速度は，それらの要求を生成したプロセスよりもずっと速い．したがって，複数個のプロセスが単一のチャネルを共有することには，意味があるのである．共有されたチャネルはすべてのプロセスからの要求を処理しているのであり，それだけ効率よく使用できる．

ソフトウェアの共有も，非能率さを減少させるためには同じように重要なことである．コンパイラのようなユーティリティ・プログラムを考えてみよう．プロセスが，プログラムを翻訳したいときには，コンパイラを自分の記憶領域に写してそれを実行する．複数個のプロセスが同時にコンパイラを使用しているときには，重複したコードを使用しているために多くの記憶領域が無駄になる．記憶の中には，コンパイラの写しを一つだけ置いておいてプロセスがこの写しを共有するならば，記憶を節約することができる．したがって，ハードウェアおよびソフトウェアにおいて，資源を共有すればシステムの効率を向上することができる．

ハードウェアとソフトウェアの資源は，共通の性質を持っている．一時には，ただ一つのプロセスの要求しか処理できないようなファイル・システムを考えてみよう．ファイル・システムはソフトウェアの資源であるけれども，多くのハードウェアの資源と同じようにして共有する．複数個のプロセスがそれを使用しようとするならば，ファイル・システムはディスク駆動装置やCPUと同じように重大な隘路となる可能性がある．物理的な装置は**物理的**（physical）または**自然な**（natural）**資源**と呼ぶ．あたかも物理的資源のように働くソフトウェアを，**論理的資源**（logical resource）と呼ぶ．論理的資源はソフトウェアで実現して，初めて存在する．たとえば，複数個のプロセスが一つのディスクを共有するときは，システムは物理的な駆動装置を共有するような論理的ディスクを多数作り出す．論理ディスクの実体は，ディスク・チャネルと相互通信をしているプロセスである．プロセスが論理ディスクを使用しているときは，自分はディスク全体を使用しているものと“考えている”．実際には，論理ディスクを持っているだけであるが，それはディスク・チャネルと**コマンド**（command）のやり取りをして，物理的装置のシミュレーションをしているのであ

る．資源を使用しているプロセスは，その資源がソフトウェアによって実現される論理的なものなのか，あるいはハードウェアで実現されている物理的なものなのかを気にかけない．プロセスの唯一の関心は，資源が期待されたとおりに働いてくれるかどうかということである．資源の共有について論ずるときは，論理的なものか物理的なものかの区別は，一般には考えない．その外的振舞は，本質的には等価だからである．

2.4.1　プロセスの同期

物理的あるいは論理的資源は，共有することができるけれども，一般に，それらは一時にはただひとつのプロセスによって使用できるだけである．一時には，一つのユーザにしか使用を許さない資源は**危険資源**（critical resource）と呼ぶ．複数個のプロセスが危険資源を使用したいと望むのならば，それらの実行過程を同期させて，高々一つのプロセスが資源を専有するようにしなければならない．一つのプロセスがその資源を使用しているならば，その資源を要求している他のすべてのプロセスは一時的に排除されて，それが解放されるまで待たなければならない．各プロセスにおいては，危険資源を使用する部分を他の部分と分離させることができる．このような部分を**危険部分**（critical section）というが，それは，**相互排除的**（mutually exclusive）性質を持っていなければならない．すなわち，一つの資源に関する危険部分は，一時には，高々一つのプロセスしか実行することができないのである．

テープ駆動装置やカード・リーダなどの多くの物理的資源は危険資源と考えられる．しかし，また，多数のプロセスによって変更される共有変数も危険資源と見做される．たとえば，AとBの２つのプロセスが，COUNT という変数を共有しているものとしよう．もしも，AとBの両方が，同時に COUNT を一つづつ増加しようとするならば，COUNT の最終値は正しくならない恐れがある．たとえば，次に示すような事象が起ったと考えよう：

1. プロセスAが，その作業変数TEMPA に COUNT の値を格納する．
2. プロセスBが，その作業変数 TEMPB に COUNT の値を格納する．
3. プロセスAが，TEMPA の値を１増加し，それを COUNT に格納する．

4.　プロセスBが，TEMPB の値を1増加し，それを COUNT に格納する．

たとえ2つのプロセスがそれぞれ COUNT の値を1づつ増加しても，COUNT の最終値は2ではなく1しか増加していない．このような望ましくない動作を避けるためには，COUNT の増加を危険部分として扱わなければならないのである．実際，複数個のプロセスが共有変数の値を増加することができるとき，誤った動作を避けるためにはその変数を危険資源として処理しなければならない．

図 2-2 に示すような2つの独立したプロセスを考えよう．各プロセスはその危険部分を次々と実行し，その後で残りの仕事を行なうものとする．危険部分は，共有の危険資源を使用する幾つかの操作をあらわしている．2つの危険部

```
parbegin
    PROCESS 1:    do    while (true);
                        begin
                            critical section 1;
                        end;
                        remainder of process 1;
                  end;
    PROCESS 2:    do    while (true);
                        begin
                            critical section 2;
                        end
                        remainder of process 2;
                  end
parend


ただし，要素：
    parbegin
        statement 1;
        statement 2;
            ⋮
        statement N
    parend
は，文1からNまで，同時に実行できることを示す．
```

図 2-2　危険部分問題

分が同時に実行されることを禁止するために，2つのプロセスの同期をとる機構が実現されていなければならない．この機構は2つの性質を持っていなければならない．第一に，1個以上のプロセスが危険部分の使用を望むならば，少なくとも一つのプロセスはいつかそこに入ることを許されなければならない．第二に，一時には，高々，一つのプロセスだけしか危険部分にいることを許してはならない．危険部分の同期の問題に対するいくつかの解については，この章の終りで議論する．

相互排除のためにお互に同期をとるほかに，プロセスは，データを交換するためにも，相互通信をしなければならない．たとえば，ユーザのプロセスはテープやディスクの装塡を要求するために，コンソールのオペレータと相互通信をしなければならない．このような型の相互通信はプロセスがメッセージを送ることを許すことにより簡単に実現できる．

プロセス PRODUCER は他のプロセス CONSUMER に対し，一群のデータを伝送するものとしよう．これらの一群のデータはメッセージと考えることができる．たとえば，PRODUCER は出力の列を生成しているユーザ・プロセスであり，CONSUMER はプリンタにその列を出力しているプロセスと考えればよい．

メッセージの伝送を実現する一つの方法は，空の**バッファ**（buffer）のプールを作り，そこでは各バッファは一つのメッセージを持つことにする．PRODUCER と CONSUMER の間には，メッセージを持ったバッファの待ち行列が存在する．PRODUCER がメッセージを伝送したいときには，バッファを待ち行列の最後尾につけ加える．CONSUMER は，待ち行列の先頭にあるバッファを取り出すことにより，メッセージを受け取る．解は簡単なものであるけれども，PRODUCER と CONSUMER とは多くの点で協力し合わなければならない．たとえば，両者は使用中と空とのバッファの数を知っておかなければならない．PRODUCER は，空のバッファが使用できるときに限り，伝送することができる．同様に，CONSUMER は待ち行列が空でないときに限り，受けとることができる．あきらかに，使用中と空のバッファの数を勘定するために，共有変数が必要となる．しかし，変数は共有しているので危険部分が必要となる．したがって，使用中と空とのバッファの勘定を変更するのは，相互排除の状態

で行なわなければならない．同様の問題がメッセージの待ち行列の処理においてもあらわれてくる．待ち行列を**結合リスト**（linked list）として実現していると仮定しよう．もしも CONSUMER が最後尾のメッセージを待ち行列から取り出しているときに，PRODUCER がちょうどメッセージを待ち行列につけ加えようとするならば，誤った結合が生じてしまう．この状況は，変数 COUNT の値が２つのプロセスによって増加させられたという例と似ている．したがって，待ち行列に対するポインタの操作も，やはり，危険部分を構成しているのである．すなわち，“簡単な”生産者－消費者の関係を実現するためには，かなりの注意が必要だということである．

PRODUCER や CONSUMER のように，ある外部の機構を通じてお互に平等に相互通信を行なうプロセスのことを**コルーチン**（coroutine）[Conway, 1963]と呼ぶ．コルーチンはサブルーチンのように，“呼んだり”，“呼ばれたり”する手続きとして関係づけられているのではない．むしろ，お互に呼び合って協力する．残念なことに，“コルーチン”という術語は文献の中で相反するような定義を与えられてきた．コルーチンに関する混乱を除くために，本書では，コルーチンの代りに“協力し合うプロセス”という概念を用いることにした．

生産者－消費者の例では，メッセージは一方向のみにしか動かない．もしも，PRODUCER がメッセージが受け取られたことを知る必要があるならば，CONSUMER は各メッセージの受信を教えてやらなければならない．このためには反対の方向へ動く返事が必要となる．システムの中に多数の生産者－消費者の対が動いているならば，空のバッファは共通のプールにおいて共有することができる．実際のシステム設計において生ずる生産者－消費者の代表的な問題としてこのような複雑な問題がある．

本節では，プロセスの相互通信に関する２つの基本的な問題を議論した．すなわち，危険部分の問題と生産者－消費者の問題である．次の２つの節では，不必要な相互干渉を避けながら有効にプロセスが相互通信できるような機構をいくつか概説し，これらの問題の解を示そう．

2.5 低レベルの同期基本命令

プロセスの**同期基本命令**（synchronization primitives）の最も基本的なもの

は，ハードウェアに密着して働いている．すなわち，記憶の相互ロック，テスト・アンド・セット，腕木システムである．これらの機構のどれを用いても，独立なプロセスの間における相互排除関係および生産者－消費者関係を実現することができる．

2.5.1 記憶の相互ロック

ハードウェアにおける相互排除は，格納操作を**非可分**（indivisible）にすることによって実現できる．すなわち，2つのプロセスが，各々単一の記憶場所へある値を格納しようとしたとき，ハードウェアは仲裁をする．すなわち，一つのプロセスには格納操作がすぐに許されるが，もう一方のプロセスは最初のプロセスが完了するまで待たなければならない．この仲裁操作は**記憶の相互ロック**（memory interlock）と呼ばれるが，これを用いれば，2個以上のプロセスの間に相互排除を実現することができる．

図 2-2 に示した危険部分の問題を，もう一度考えてみよう．解は，一時には，高々一個のプロセスしか危険部分に入ることができないこと，および，一個以上のプロセスが危険部分に入ることを望んだときは，必ずいつかは一つのプロセスが入れることを保証していなければならない．2つのプロセスの速度は独立であるから，各プロセスがもう一方のプロセスに対してどんなに速く実行されても，前述の条件は成立しなければならない．次の例は試行的な解であるが，後者の条件を満足することはできない．

図 2-3 において，各プロセスは一つのスイッチを持っているが，それは自分が危険部分にいるときに“**真**”の値，そうでないときに“**偽**”の値をとる．危険部分に入る前に，プロセスは安全に入ることを保証するために，相手のプロセスのスイッチを調べる．それから自分自身のスイッチを *on* にし，危険部分を使用する．しかしながら，この“解”の安全性は，幻想であることがわかる．プロセス2が危険部分にいるときに，プロセス1がループ1にいるとしよう．プロセス2はその危険部分の処理を終了し，スイッチ2を**偽**にする．この結果，プロセス1はループ1の実行から解放される．実行速度は任意であるから，プロセス2がプロセス1よりもずっと速く実行されると仮定しよう．プロセス2が非常に速いので，実際，スイッチ2の値が**偽**であることをプロセス1が見つ

```
Begin boolean switch 1, switch 2;
      switch 1 := false;
      switch 2 := false;
      parbegin
         process 1 :     do    while (true);
            loop 1 :             do while (switch 2);
                                 end;
                                 switch 1 := true;
                                   /* critical section 1 */
                                 switch 1 := false;
                                   /* remainder of process 1 */
                         end;
         process 2 :     do    while (true);
            loop 2 :             do while (switch 1);
                                 end;
                                 switch 2 := true;
                                   /* critical section 2 */
                                 switch 2 := false;
                                   /* remainder of process 2 */
                         end
      parend
end
```

図 2-3 危険部分の誤った解

けたが，スイッチ1を未だ**真**にしてしまわないうちに，プロセス2がその繰り返しの残りの部分を急いで進めてしまい，ループ2を済ませてしまう（スイッチ1が，未だ**偽**であるから）. 両方のプロセスは“安全に”進行してしまい，お互のスイッチを**真**にセットし危険部分を実行してしまう．任意の速度で処理されることを仮定した結果，両方のプロセスがそれぞれの危険部分を同時に実行してしまうような事象の列が見つかってしまう．したがって，**図 2-3** は，危険部分の問題の解ではない．

読者はこのプログラムを修正して，危険部分の解の条件を満足するようにしたいと思うだろう．それは，そう簡単ではない演習問題である．Dekker による最初の正解を**図 2-4** に示す（R. C. Holt が，GO TO 文を使わずに書き直した）. この解では，プロセス1が危険部分に入りたいときには C1=1 が成り立ち，プロセス2が危険部分に入りたいときには C2=1 が成り立つ．また，“turn”は両方のプロセスが危険部分の使用を望んでいるときに，どちらが入ってよいかを指定する変数である．Dekker のアルゴリズムが如何に巧妙に出来

```
Begin integer  c1 ,c2, turn ;
               c1 := 0 ;
               c2 := 0 ;
               turn := 1 ;
    parbegin
       process 1 :  begin c1 := 1 ;
                          do while (c2 = 1) ;
                             if turn = 2
                             then  begin c1 := 0 ;
                                         do while (turn = 2) ;
                                         end ;
                                         c1 := 1
                                   end
                          end ;
                          critical section for process 1 ;
                          c1 := 0 ;
                          turn := 2 ;
                          remainder of process 1
                    end ;
       process 2 :  begin c2 := 1 ;
                          do while (c1 = 1) ;
                             if turn = 1
                             then  begin c2 := 0 ;
                                         do while (turn = 1) ;
                                         end ;
                                         c2 := 1
                                   end
                          end ;
                          critical section for process 2 ;
                          c2 := 0 ;
                          turn := 1 ;
                          remainder of process 2
                    end
    parend
end
```

図 2-4 Dekker のアルゴリズム

ているかを知るために，プロセサの相対的な速度をいろいろ変えて手で実行してみるとよいだろう．この解は正しく動くと証明されている [Dijkstra, 1968a] が，危険資源を競い合っている任意個のプロセスの場合に対しても，一般化さ

れている [Dijkstra, 1966 ; Knuth, 1966]†.

2.5.2 テスト・アンド・セット

相互排除を実現するために，多くの機械では，単純な非可分の格納操作よりもずっと使い易く効率的なハードウェアの**命令** (instruction) を用意している．そのような命令の一つが，"テスト・アンド・セット" である．

ハードウェア命令の**テスト・アンド・セット** (test and set) は，記憶の相互ロックに比べて，危険部分の問題の解を飛躍的に簡単にすることができる．テスト・アンド・セットは，2つのパラメータ LOCAL と COMMON とを持って呼び出される．この命令は COMMON の値をとり出し，それを LOCAL に割り当てる．そして COMMON の値を1にセットする．この操作の重要な性質はそれが非可分であるということである．プロセスがテスト・アンド・セットの命令を実行しているときは，その命令の開始から終了までの間は他のどんな操作も行なうことができない．

変数 COMMON はある危険資源に関して同期をとろうとしているプロセス全体が共有している．各プロセスは自分個人用の作業変数 LOCAL を持っている．IBM/360 におけるテスト・アンド・セットの実施では，機械の**条件コード** (condition code) が作業変数として働いている．COMMON=1 が成り立っているときは，危険部分のなかにプロセスがいることを示している．最初は，COMMON=0 としておく．**図2-5**はテスト・アンド・セット命令を用いた2つのプロセスによる相互排除問題の解を示している．この解においては，機械は記憶の相互ロックをそなえているものと仮定する．すなわち，COMMON:=0 という命令は非可分に行なわれるものとする．

記憶の相互ロックとテスト・アンド・セットはそれぞれ相互排除を実現するための能力を持っているが，それらは両方とも非能率的である．一つのプロセスがその危険部分を実行しているときは，他のどんなプロセスもループ内にとどまり，先へ進む許しのでるまで待っている．このあてのないループを**ビジー待ち** (busy waiting) と呼ぶが，それは処理装置の時間を無駄に費す．危険資

† 訳注. Dijkstra の解には，見かけ上のデッドロック (effective deadlock) が存在することを Knuth が指摘した．したがって，完全に正しいものは，Knuth の解 [1966] である．

```
Begin     integer      COMMON ;
          COMMON := 0 ;
  parbegin
    process 1 :   begin  integer LOCAL 1 ;
                         do while (true) ;
                            LOCAL 1 := 1 ;
                            do while (LOCAL 1 = 1) ;
                               Test and set (LOCAL 1, COMMON)
                            end ;
                            critical section 1 ;
                            COMMON := 0 ;
                            remainder of process 1
                         end
                  end ;
    process 2 :   begin  integer LOCAL 2 ;
                         do while (true) ;
                            LOCAL 2 := 1 ;
                            do while (LOCAL 2 = 1) ;
                               Test and set (LOCAL 2, COMMON) ;
                            end ;
                            critical section 2 ;
                            COMMON := 0 ;
                            remainder of process 2
                         end
                  end
  parend
end
```

図 2-5 テスト・アンド・セットを用いた相互排除問題

源を用いているプロセスがそれを放棄しようと決心するまでは，それを待っている他のすべての プロセスは“寝てしまい”，実行に用いている 処理装置を放棄することが望ましい．危険部分が再び解放されたときは，“寝ている”プロセスが１つ起こされてその処理装置を返して貰い，危険資源の使用を許してもらう．ビジー待ちを用いないで待つことができるひとつの簡単な同期の機構は“腕木システム”である．

2.5.3 腕木システム

腕木（semaphore）はときには**汎用腕木**（general semaphore）ともいうが，それはＰおよびＶ命令［Dijkstra, 1968a, b］によってのみ値が変更される整数

型変数である．S を腕木としよう，プロセスが $P(S)$ を実行すると，S の値が1減らされ，

1. $S \geqq 0$ ならば，そのプロセスは実行を続ける．

2. $S<0$ ならば，そのプロセスは停止し，S に対して用意された待ち行列に入る．そして他のプロセスが発した $V(S)$ 命令により解放されるまで待つ．

プロセスが $V(S)$ を実行すると，S の値が1増加させられ，

1. $S>0$ ならば，プロセスは実行を続ける．

2. $S \leqq 0$ ならば，1つのプロセスを待ち行列から取り出し，実行を継続することを許す．$V(S)$ 命令を実行したプロセスも実行を継続できる．

また，PおよびV命令は非可分操作である．与えられた腕木に対しては，一時には，ただ一つのプロセスだけが，PまたはV命令を実行することができる．したがって，$S=1$ でかつ2つのプロセスが同時に $P(S)$ を実行しようとすると，一方のプロセスだけが実行を続けることを許される．もう一方のプロセスは停止し，S の待ち行列に入る．

第2.3節で，プロセスを実現している核心を定義したことをもう一度考えてみよう．核心の部分はプロセスに処理装置を割り当てたり，解除したりする責任を持っている．核心はプロセスを開始させたり停止させたりする能力を持っているので，PおよびV命令を核心において実現することができる．非可分性を保証するために，与えられた腕木に対して，一時には，ただ，一つのPまたはV命令だけを処理するという条件を核心は守らなければならない．PおよびV命令の能力を用いれば，危険部分の問題はビジー待ちを用いないでも解くことができる．

最大値が1にしかならないような腕木を**2値腕木**（binary semaphore）と呼ぶ．2値腕木 S を用いると，危険部分の前後を $P(S)$ および $V(S)$ 命令で囲むことだけで，プロセスは，相互排除を実現することができる(**図2-6**参照)．もしも，2つのプロセスがその危険部分での同期をとるために S を用いているならば，腕木の定義により，S は 1，0，-1 のうちのどれかの値をとる．$S=1$

```
Begin    integer free;
         free := 1;
         parbegin
            process 1 :  begin
                            do while (true);
                               start of process 1;
                               P (free);
                                  critical section 1;
                               V (free);
                               remainder of process 1
                            end
                         end;
            process 2 :  begin
                            do while (true);
                               start of process 2;
                               P (free);
                                  critical section 2;
                               V (free);
                               remainder of process 2
                            end
                         end
         parend
end
```

図 2-6 腕木を用いた危険部分の問題

ならば，どちらのプロセスも危険部分を実行してはいない．$S=0$ ならば，どちらか一方のプロセスがその危険部分を実行している，$S=-1$ ならば，一方のプロセスがその危険部分を実行中であり，もう一方のプロセスがSの待ち行列内にいて，危険部分に入ることを許されるまで待っている．2値腕木を用いたこの簡潔な解はもっと多数のプロセスが存在しても，同様にうまく働く．Dekker のアルゴリズムは，2つ以上のプロセスがあるときは非常に複雑になるけれども，腕木による解は自明である．この点が，腕木システムを用いる主要な利点である．

汎用腕木システムを用いると，生産者－消費者問題を簡単に解くことができる（**図 2-7** 参照）．腕木 AVAIL はバッファのプール内の空のバッファの数を

```
Begin   integer avail, full, mutex ;
        avail := number of empty buffers ;
        full := 0 ;
        mutex := 1 ;
        parbegin
          producer :  begin
                          do while (true) ;
                             prepare message ;
                             P (avail) ;
                             P (mutex) ;
                             send message ;
                             V (full) ;
                             V (mutex)
                          end
                      end ;
          consumer :  begin
                          do while (true) ;
                             P (full) ;
                             P (mutex) ;
                             receive message ;
                             V (avail) ;
                             V (mutex) ;
                             process message
                          end
                      end
        parend
end
```

図 2-7　腕木を用いた生産者－消費者問題の解

あらわす．腕木 FULL は消費者に送られた使用中のバッファの数をあらわす．腕木 MUTEX (MUTual EXclusion――相互排除）は2値腕木であり，一時にはただ一つのプロセスだけがバッファのポインタを操作することを保証するために用いる．

汎用腕木は資源を管理するための簡潔な手段を与えている．生産者－消費者問題の解においては，汎用腕木は使用中および空のバッファの数をカウントするために用いている．これらはテープ駆動装置，ディスク駆動装置，タイプライタ端末などを割り当てるときに同様にしてうまく使用することができる．DISK を，使用できるディスク駆動装置の数とする．プロセスは P(DISK) を実行することによりディスクを要求する．ディスクは V(DISK) を実行するこ

とにより解放される．一つのディスク駆動装置が，２つのプロセスに割り当てられるということはない．DISK は，一時には，１だけ減らされるからである．

腕木は，また，同期のためにも用いることができる．たとえば，プロセス１はプロセス２に対してある仕事をして欲しいと要求するとしよう．プロセス２が完了するまでは，プロセス１は何もすることがない．そこで P(WAIT) を実行するが，WAIT は初期値が０であるような腕木である．プロセス２がその仕事を完了すると，V(WAIT) を実行してプロセス１に知らせる．このような腕木の使い方は危険部分の問題と考え方が異っているわけではない．何故ならば，プロセス２は腕木 WAIT によって保護されている資源と考えることが出来るからである．しかしながら，腕木を同期や資源の管理のために使用していると，腕木の待ち行列のスケジューリングに複雑な問題が生じてくる．

腕木 S が負の値になると，S に対して用意された待ち行列内に１個以上のプロセスが存在する．次の V(S) 命令が実行されると，核心は待ち行列から取り除くべきプロセスを選ばなければならない．核心は**先来先処理**（first-come-first-served；略して **FCFS**）方針に従い，待ち行列から最も長く待っていたプロセスを取り除く．あるいは高順位のものが先になるようなもっと複雑なスケジューリング・アルゴリズムを用いて，次に取り除かれるプロセスを決めるかもしれない．スケジューリングの方針は腕木を使用する目的に基づいて選ばれなければならない．

たとえば，危険部分問題に対する腕木を用いた解を考えてみよう．危険部分が非常に長いとそれを共有しているすべてのプロセスに対する隘路にすぐなってしまう．したがって，危険部分は，通常，非常に短かくなければならない．これは危険部分に入るためには，どんなプロセスもそんなに長く待ってはならないことを意味する．プロセスは腕木の待ち行列の中では，どのみち，長い時間を過すことはないので，最も簡単なスケジューリングの方針，たとえば，FCFS を使用すればよろしい．しかしながら，資源の管理や同期においてはもっと複雑な法則が必要になることが多い．待ち時間が結構重要な点になるからである．

ディスク駆動装置を割り当てるために用いる腕木 DISK を思い出してみよう．すべてのディスクが使用され，別にいくつかのプロセスが駆動装置を要求

しているものと仮定しよう．すなわち，DISK<0 が成り立っている．次のプロセスが V(DISK) によって一つのディスクを解放したとき，どの待ちプロセスを解放するかということは，重要な問題になるであろう．たとえば，一つのプロセスは，すでに3つのテープ駆動装置を持っているかもしれない．このプロセスは高価な資源を何も持っていないようなプロセスよりも先に，新しく解放されたディスクを受け取るべきである．この場合，優先順位に基づいたスケジューリング方法が望ましい．プロセスの優先順位はそのプロセスが支配している高価な資源の数によって，動的に決めることができる．この例からもわかるように，単一のスケジューリング法則は腕木の待ち行列を実現するためには，一般には充分ではない．

腕木システムは，一般に，オペレーティング・システムにおいて必要となる同期や相互通信を実現するためには，充分な機能を備えていると言える．THE 多重プログラミング・システム（第8章参照）および Venus オペレーティング・システム（第10章参照）は，両方とも基本的な通信機構として，腕木システムを用いている．しかしながら，腕木システムは非常に基本的なものであるので，ときには全く不便なことがある．たとえば，並行的プロセスの集団において，凝ったメッセージの交換の機構を腕木システムで実行するとかなり複雑なものとなる．もっと洗練された通信手段を用いて，共通の同期問題のもっと簡単な解を得ることができる．これらの洗練された通信手段のいくつかを，次の節で紹介する．

2.6 高レベルの同期基本命令

これまで議論してきた同期の機構は論理的には完全であり，非常に能力も高かったが，それらを用いるには多くの危険性が伴っている．それらが基本的であるという性質のために非常に複雑な構成になってしまい，理解しにくく，また，少量の変更に対して大きく左右されてしまう．危険部分や生産者－消費者の問題の解を少し変更すると，破滅的な結果になってしまう．本節で述べる同期基本命令を用いれば，理解し易い解を与えてくれる．それゆえ，プログラマは自分のコーディングに高度な信頼をおくことができる．それらは精巧な通信機構を与えているが，高レベルの基本命令を実現するためには入念さと手間と

が必要となる．しかしながら，その労力はプログラムの誤りの確率が減るということにより相殺されるということを，いくつかの証拠事実が示している．

2.6.1 郵便受け

郵便受けの名前は，手紙を送るときの伝統的な機構から来ている．プロセスP1 がプロセス P2 と通信をするときは，P1 は，2つのプロセスを連絡するための郵便受けを作ることを要求し，メッセージが両者の間で伝送できるようにする．P2 と通信をするためには，P1 は郵便受けにメッセージを置くだけでよい．P2 は任意の時間にそれを取り出す．郵便受けは，P2 が要求したときは，それ以降，どこかで P2 がメッセージを受けとることを保証している．

郵便受け（mail box）は，実際は操作の法則を付随したデータ構造である．論理的には，郵便受けを記述する**ヘッダ**（header）項目とメッセージが置かれるいくつかの**スロット**（slot）とから成り立っている．郵便受けが作られたときには，スロットの大きさとスロットの数とは一般に決められている．操作の法則は，郵便受けの複雑さによって異なる．最も簡単な場合は，メッセージは一方向へしか転送されない．プロセス P1 は空いているスロットのある間は，メッセージを送ることができる．すべてのスロットが使用中の場合には，P1 は待っているかまたはしばらく実行を続けてから，メッセージの転送を再度試みることができる．同様に，P2 は使用中のスロットのある間は，メッセージを受け取ることができる．メッセージのない場合は，待っているか，または実行を続けることができる．もっと凝った通信機構を持つためには，この様な簡単な郵便受けの設計をいろいろの方式を用いて複雑にすることができる．

2.6.1.1 二方向の通信

P2 がメッセージを受け取りある動作をしたという返事をP1が受け取りたいと思うことが，ときどきあるであろう．これは郵便受けにおいて，両方向への通信を許すことによって可能となる．この場合，各スロットには P1 からのメッセージかまたはP2からの返事かどちらかが入っている．この方式における問題点は，P2が返事を送るよりも速くP1がメッセージを送り続けることである．P1 はすべてのスロットを使用してしまい，P2 からの返事に対する余地を残さなくなってしまうだろう．このような状況は，返事を送るスロットとして，元

のメッセージを送ったスロットと同じものを使うことにより解決できる．メッセージが送られたときには，それに対する返事が来るまでそのスロットを確保しておく．この条件を守れば，P2 が返事を送る時間がなくなるほどたくさんのメッセージをP1が送り出すという事態はなくなる．

2.6.1.2 多入力の郵便受け

重要なユーティリティ，たとえば，ファイル・システムや料金計算システムを実行する一つのプロセスと，多数のプロセスとが相互通信をしたいことがしばしばある．このような状況は，**デモン問題**（demon problem）と呼ばれることがある．単一の入力と出力を持った郵便受けを用いると，この問題の処理は非常に非能率的になる．各プロセスはユーティリティ・プロセスと通信するために，それぞれ異った郵便受けを持たなければならない．さらにユーティリティ・プロセスは要求を知るために，これらの郵便受けを調べなければならない．多入力の郵便受けならば，複数個のプロセスがメッセージを置くことができる．それらのメッセージは何らかの優先順位に従って受け取られるであろう．当然のことではあるが，多入力郵便受けは単一入力のものに比べて，実現するのはずっと難しい．しかしながら，デモン問題を解く場合に多入力郵便受けを用いれば，たくさんの郵便受けを用いる場合に比べて，能率がずっとよくなる．多出力郵便受けも同様にして定義することができる．

2.6.1.3 ポ ー ト

プロセスはメッセージを送るためには，郵便受けの名前を知らなければならない．これは，ときには，不便なことがある．たとえば，いくつかのディスク駆動装置を持つシステムを考えてみよう．そこでは各ディスクに対し駆動プロセスが与えられ，チャネルに対する読み書きの要求を伝えている．プロセスがある特定の駆動装置を使用したいときは，二方向の郵便受けを駆動プロセスにつなぎ，ディスクへの命令をその郵便受けに送る．駆動プロセスは“書き出し命令が終了した”とか，“読み込みを要求されたレコードが，ここにあります”というようなメッセージを返してくる．明らかに，駆動プロセスはどのプロセスが処理を要求しているかという点については，無関心である．何故なら，読み書きの要求はどのプロセスが送ってきても同じだからである．プロセスが使用している郵便受けの名前を知らなくても済むようにするためには，プロセス

はポートを用いて通信すればよい．**ポート**（port）はプロセスを郵便受けに接続するコネクタである．郵便受けがポートに接続された後は，プロセスはメッセージを送るために，ポートの名前を教えるだけでよい．前述した例においては，ディスクの駆動プロセスはポートがどのプロセスの郵便受けに接続しているかということは全く無視して，一つのポートを介してメッセージや返事を転送するのである．郵便受けの名前は，それが物理的に2つのポートに接続しているときだけ必要となる．

郵便受けの動作は，例を示すことによってよく理解できる．次に示す郵便受けシステムは，（実現はされなかったが）SUE システムに対して設計された［Atwood ほか，1972］．

2.6.1.4 SUE における郵便受けシステム

プロセスが，2.3節に示した核心によって実現されていると仮定しよう．SUEシステムの設計では，ポートによって接続する二方向郵便受けを用いている．プロセスは2つの型のポートを持っている．すなわち**入力ポート**（input port）と**出力ポート**（output port）である．各郵便受けは一つの入力ポートと一つの出力ポートに接続される．出力ポートを介して送られるメッセージを**質問**（query）と呼び，入力ポートを介して送られるメッセージを**返答**(reply)と呼ぶ．郵便受けの各スロットは，次の4つの状態のうちのどれかになっている．すなわち，質問待ち，質問在り，返答待ち，返答在りの4つである．メッセージを転送するためにシステムが用意した命令は**郵便受けの作成**（create mailbox），**郵便受けの消滅**（destroy mailbox），**接続**（connect），**送信**（send），**受信**（receive）である．この設計においては，接続していないポートは“ダミーの郵便受け”に，接続していない郵便受けは“ダミーのポート”に接続していると考える．

郵便受けの作成命令は，指定したスロットの大きさとスロットの数を持った郵便受けを作り出す．郵便受けの持ち主すなわちそれを作り出したプロセスは，ヘッダとスロットのための領域を与えなければならない．郵便受けを作ったときは，それは2つのダミーのポートへ接続される．

いったん作られた郵便受けは，接続命令を用いてポートに接続される．接続の切り離しは，ダミーのポートやダミーの郵便受けに接続することによって行

なわれる．

メッセージは指定したポートを介して，送信命令により転送される．指定した出力（または入力）ポートに接続している郵便受けに対して，質問待ち（または返答待ち）の状態のスロットが使用できるかどうかのチェックをする．もし何か使用できれば，メッセージがスロットの中に写され，そのスロットは質問在り（または返答在り）という印がつけられる．使用できるスロットがないときには，メッセージは転送されず呼び出したプロセスに警告がなされる．この場合，呼び出したプロセスの状態には影響を与えない．すなわち，そのプロセスは停止されない．

メッセージを受け取るためには，いくつかの入力（または出力）ポートをパラメータとして指定しながら受信命令を呼び出す．各ポートに接続している郵便受けに対して，質問在り（または返答在り）の状態のスロットが使用できるかどうかチェックする．もしも，どれかのポートがこの条件を満しているならば，メッセージをこのスロットから取り出し，呼び出したプロセスが指定した領域に写す．呼び出すプロセスはメッセージがないときは停止するかまたは単にメッセージがないということだけ知り，実行は継続するように要求することができる．

郵便受けはその持主によって，消滅命令を用いていつでも消滅させてよい．その郵便受けが未だ2つのダミー・ポートに接続されていないならば，それはダミーの郵便受けと置きかわる．郵便受けが占めていた領域は，その持主のプロセスに返却される．

2.6.1.5 注 意 事 項

これまでに述べてきたすべての通信機構においては，通信したいと望んでいる相手のプロセスを，一意的に識別できることが必要であった．郵便受けシステムにおいては，郵便受けを接続するためにプロセスを識別することが必要である．プロセスの名前を知ることは，システムにおいてはいろいろの能力の基本となることである．それによりプロセスは他のプロセスのポートや郵便受けの接続に干渉し，また恐らく，それらのメッセージも読むことができるであろう．意地の悪いプロセスが大恐慌をひき起さないようにするために，システムはプロセスの名前や郵便受けのスロットを許可しない使用から保護するための

規則を守らせなければならない．そのような保護の規則を守らせる技術については，第7章で詳述する．

郵便受けシステムはメッセージの転送には便利であるけれども，それらは危険部分の同期に対しては，簡単には使えない．この同期は危険部分の使用を許可することの責任を持つ特別なプロセスを用意することにより実現できる．プロセスは危険部分に入りたいときに，その特別なプロセスにメッセージを送る．それは返事を受け取ったら，実行を続けてよろしい．特別なプロセスは，危険部分は同時にはただ一つのプロセスだけしか使用を許さないように管理している．同期に対するこのような“特別のプロセスを用いた解”から，モニタの概念が引き出されるのである．

2.6.2 Hoare のモニタ

モニタ (monitor) はたくさんのプロセスの間で共有される手続きとデータ構造の集まりであるが，それらは，一時には，一つのプロセスしか使用できない [Hoare および Perrot, 1972；Dijkstra, 1971]．モニタの概念は一つの鍵しかない部屋の考え方に似ている．もしも，プロセスが部屋を使用することを示し鍵がドアにぶらさがっているならば，そのプロセスは部屋を開けて中に入り，モニタの手続きのうちの一つを使ってよろしい．もしも，鍵がドアにぶらさがっていなければ，そのプロセスは現在のユーザが部屋を去り，鍵を放すまで待たなければならない．さらに，部屋に永久に残っていることは誰にも許されない．

たとえば，スケジューラのプログラムによって割り当てられる資源を考えてみよう．プロセスが資源を数個得ようと考えるときは，いつもスケジューラを呼び出さなければならない．スケジューラの手続きは，すべてのプロセスが共有している．すなわち，どのプロセスもいつでもスケジューラを呼び出してもよい．しかしながら，スケジューラは一時にはただ一つのプロセスに対する処理しかできない．このスケジューラは，モニタの一つの例である．

モニタは，それを呼び出したプロセスを遅らせなければならないことがときどきある．たとえば，すでに使用中の資源への要求を，あるプロセスがモニタに示したときには，モニタはその資源が使用可能になるまでそのプロセスを停止しておかなければならない．プロセスは**待ち** (wait) **命令**と**信号** (signal) 命

命を用いて，**停止**（blocked）したり**停止解除**（unblocked）になったりする．モニタが待ち命令とともにプロセスを停止するときは，プロセスが実行を再開するための**条件**（condition），たとえば，“資源は使用中でない”を指定しておかなければならない．その条件が成立するようになったときには，モニタはその条件に対する信号命令を出す．その条件を待っているプロセスが存在すれば，そのうちの一つを呼び起しその実行を続けることを許す．すなわち，それを停止していた待ち命令のちょうど一つ後の命令が，次に実行されるのである．もしも，その条件を待っているものが一つも存在しなければ，信号命令は何の効果ももたない．

モニタは固定した全域的データ構造を持った一つ以上のリエントラント手続きを持っている．モニタを最初に呼び出したときは，その変数の初期設定を行なう．それ以後の呼び出しにおいては，前の呼び出しで残っている変数の値を使用する．モニタは，部屋のように受動的なものであることを強調しておかなければならない．それはプロセスではないのである．モニタはプロセスがその処理を使用しようとするときだけ，生きているのである．モニタのコードのユニークな特徴は，一時には，そのコードをただ一つのプロセスだけが実行できるということである．

モニタの使用法を**図 2-8** に示す．そこでは腕木Sに対して，PおよびV命令を実現している．**条件名．待ち**（condition name. wait）および**条件名．信号**（condition name. signal）という書き方は，条件“**条件名**（condition-name)”に

```
binarysemaphore : monitor ;
   begin integer s;
         condition semaphorepositive ;
         procedure P;
            begin if s < 1 then semaphorepositive.wait ;
                  s := s - 1
            end ;
         procedure V ;
            begin s := s + 1 ;
                  if s = 1   then semaphorepositive.signal
            end ;
         s := 1
end
```

図 2-8 Hoare のモニタを用いたPおよびV命令の実現

対する待ちおよび信号命令を示している．PおよびV命令は binarysemaphore. Pおよび binarysemaphore. V という書き方で呼び出される．条件“semaphore-positive”は，停止しているプロセスが実行を継続してよい時を示すために用いている．**図 2-9** では，2つのプロセスがモニタ binarysemaphore を用いて，危険部分に対する相互排除を行なっている．使用すべき腕木を示すパラメータをモニタ・コールで用いるように修正すれば，複数個の腕木を扱うようにすることができる．

モニタの意味から判断すると，条件を待っているプロセスが少なくとも一つ存在するならば，その条件に対する信号命令と一つの待ちプロセスの実行の開始との間には，他のどんなプロセスの呼び出しも介在しないという保証がなされている．したがって，**図 2-8** においては，Pにおける待ち命令の後でSの値を再度調べる必要はない．信号命令により $S=1$ となっており，一つのプロセスが解放される前に他のプロセスがSの値を変更することはできない．また一つ以上のプロセスが待っているのであるから，各条件に対して，プロセスの待ち行列を用意している．一番長く待っているプロセスを解放するという単純な

```
Begin
   parbegin
      P1 :  begin
               do while (true) ;
                  call binarysemaphore.P ;
                     critical section for P1 ;
                  call binarysemaphore.V ;
                     remainder of P1
               end
            end ;
      P2 :  begin
               do while (true) ;
                  call binarysemaphore.P ;
                     critical section for P2 ;
                  call binarysemaphore.V ;
                     remainder of P2
               end
            end
   parend
end
```

図 2-9　モニタ・コールを用いた相互排除

待ち行列方式を用いれば，どのプロセスも永久に待たされてしまうことはない．

基本的な同期および通信機構としてモニタを用いれば，プロセスはデータを陽に共有する必要はない．むしろ，共有変数はモニタの内部で使用されているのである．この条件を守れば，危険部分は自動的に消滅される．一時には，ただ一つのプロセスしかモニタを使用することができないから，2つのプロセスが同時に共有変数を使用するということはなくなる．さらに，プロセス自身の内部では変数を共有することはできないから，システムがコンパイルされるときに誤った共有を見つけ出すことができる．

モニタは，腕木システムにくらべて処理能力がふえているわけではないけれども，より基本的な同期機構にくらべて，いくつかの利点を持っている．第一に，モニタは非常に柔軟性に富んでいる．腕木システムのほかに多くの同期用命令がモニタとして実現できる．たとえば，前節に述べた郵便受け機構はモニタとして簡単にプログラムできる．また，すべての共有変数をモニタの内部に置くことにより同期するプロセスの構成が明瞭になる．複雑なプロセスの相互作用をわかり易い方法で同期させることができる．モニタは，また，危険部分のコードをプロセスに共有させることができる．各プロセスが全く同じ方法で共有資源を使用するならば，ただ一つのモニタの手続が必要になるだけである．これは，腕木システムとは対照的である．そこでは，各プロセスは危険部分の写しを自分で持っていたのである．モニタは腕木システムにくらべて能率は少し悪くなっているけれども，複雑さや理解し易さに関しては，非常によくなっていると言えよう．

SUE システムでは，同期機構として Hoare のモニタの改訂版である**ファシリティ**（facility）を実現してきた．ファシリティはモニタの機能を行なうプロセスである．モニタにくらべて，ファシリティはある意味ではもっと柔軟性を持っているけれども，プロセスに余分なデータ構造を持たせなければならないので能率が落ちる．ファシリティについては，第10章の SUE システムの記述のところで，詳しく述べる．

本章で述べた通信用基本命令は，多くの実在する機構の中から少数のものを選んで紹介したに過ぎない．ある一つのシステムに対する基本命令の選択は，効率に影響してくるハードウェアの性質によって大きく左右されるであろう．

```
P1 :                        P2 :
         ⋮                           ⋮
    P (READER) ;               P (PRINTER) ;
    P (PRINTER) ;              P (READER) ;
         ⋮                           ⋮
```

図 2-10　2つの並列プロセス

しかしながら，大抵の同期機構は，概念的には腕木システム，郵便受けシステム，モニタのどれかに非常に似ているといえる．

2.7 デッドロック

プロセスがうまく協力して働くことのできるような同期の方法を，いくつか紹介してきた．腕木システムのような通信基本命令を用いれば，プロセスはお互に停止したり停止を解除したりして，その操作の同期をとることができる．しかし同期基本命令を注意して使用しないと，予想もしなかったような問題が生じてくる．たとえば，P1 と P2 を2つの並列プロセスとしよう（**図 2-10**）．READER と PRINTER は初期値が0であるような2値腕木である．P1 が P(READER) を実行し，P2 が P(PRINTER) を実行したとすると，両方のプロセスはそれぞれ2番目のP命令で停止してしまう．さらに，どちらも2番目のP命令を通過できる望みはないので，両者とも永久に停ってしまう．お互に相手を永久に待ち続けるような状況を**デッドロック**（deadlock）と呼ぶ．

デッドロックの概念は第2.2節で示したものと同じような簡単なモデルを用いて形式化されている［Holt, 1972］．システムを**状態**（state）の集合と**プロセス**（process）の集合としよう．そこでは，各プロセスは状態から状態への写像を行なう関数と考える．プロセスがある状態で実行できないとき，その状態で**停止している**（blocked）という．プロセスはある状態，およびそこからシステムが到達できる将来のすべての状態において停止しているとき，その状態において**デッドロックしている**（deadlocked）という．どのプロセスもある状態をデッドロック状態に写像できないとき，その状態は**安全**（safe）であるという．

多くの異った種類の資源を持つシステムにおいては，デッドロックは難しい問題である．注意深く資源の管理を行なわないと，**図 2-10** に示すように2つのプロセスがお互に他方が要求している資源を保持してそれを解放したがらないという状況になってしまう．明確に定義されていようといまいと，すべてのシステムはデッドロックに関する方針を持っている．基本的には3つの方針が考えられる．すなわち，防止，自動検出およびオペレータによる検出である．

防止方式では，システムは安全でない状態に到達することは許さないという基準を守る．したがって，プロセスがデッドロックを引き起すような要求をするときは，システムは安全でない状態を避けるような処理を行なう．すなわち，要求を認めないか，またはデッドロックの可能性を除くために他のプロセスから資源をとり上げてしまう．この方法は，常にデッドロックを避けることができるという利点をもっている．しかしながら，安全でない状態へ到達するのを防ぐために，使用可能な資源への要求もときどき拒否されてしまうので，資源は使用されないままにほうっておかれることがしばしばある．また，防止アルゴリズムそれ自身，大きなオーバーヘッドになってしまう．

自動検出の方式では，デッドロック状態になることは認めるが，それが生じたときは，プログラムによりそれらを検出する．次に，システムはどれかのプロセスの資源を取り上げて他のプロセスが実行を継続できるようにする．前述したリーダ／プリンタの例では，システムはデッドロックの状態になるまではうっておく．しかし，システムはそこで P2 のプリンタを取り上げて，それを P1 に割り当てる．P2 は相変らず停止されているが，2つのプロセスはもはやデッドロックの状態になってはいない．デッドロックの回復は，自動的に行なうかオペレータの下で行なうか，どちらでも可能である．一般に，自動検出の方式では，防止の方式に比べて資源の使用効率があがる．前者は安全でない場合でも，資源を割り当ててしまうからである．デッドロックの生ずる回数が充分少なければ，資源の使用効率の増加による利益はデッドロックの回復のための労力を充分補って余りある．

三番目の方式では，デッドロックの生ずる回数は非常に少ないので，それについて心配する必要はないという観点に立っている．デッドロックが生じてオペレータがそれを検出したときは，システムは単に再開始するだけである．デ

ッドロックの検出のアルゴリズムに費す無駄を省いている．しかし，デッドロックが生じたときは，かなりの代価を支払わなければならない．オペレータはデッドロックの生じたことを，しばらくの間気が付かないかもしれない．オペレータが気が付いたときは，一般に，無駄に消費された時間ということから考えると，システムの再開始はきわめてもったいないものである．

3つの方式の中では，防止の問題が恐らく一番関心を持たれているだろう．防止アルゴリズムの例を次節に示す．

2.7.1 デッドロックの防止アルゴリズム

テープ駆動装置の使用を競いあっているプロセスのいるシステムを考えよう．必要としているテープ駆動装置が使用できるならば，プロセスは有限の時間の間，実行を続ける．各プロセスはテープ駆動装置を要求する前に，将来，同時に必要とする数の最大値を指定しなければならない．テープ駆動装置の管理システムは，要求した割り当てがデッドロックを生ずる可能性のない限り，テープ駆動装置への要求を認める．すなわち，すべてのプロセスが完了することができるような実行過程がなければならない．たとえば，3個のプロセスと10個のテープ駆動装置とから成り立つシステムを考えてみよう．各プロセスには，その最大の要求数，現在割り当てられているテープ駆動装置の数，残りの要求数の情報が与えられている．**図 2-11a** に，代表的なシステムの状態を示す．デッドロックに対する最悪の場合は，各プロセスが，それぞれ残りの要求をすべて要求することである．すべてのプロセスがこのような状況のもとで完了するならば，システムは**この状態で安全**であるという．**図 2-11a** には安全な状態が

(a)

プロセス名	最大要求	割り当て	要求
A	4	2	2
B	6	3	3
C	8	2	6

(b)

プロセス名	最大要求	割り当て	要求
A	4	2	2
B	6	3	3
C	8	4	4

図 2-11 システムの割り当て状態

示してある．さて，プロセスCが，あと2個，テープ駆動装置を要求したとしよう．もしも，Cの要求が認められかつすべてのプロセスが残りの要求を主張したならば，システムはデッドロックに陥入ってしまう．したがって，Cの要求を認めるのは安全ではない．

一般に，ある要求を認めることが安全でない状態をひきおこすかどうかを検出するようなアルゴリズムを得ることは可能である．一つの検出手続きは“**銀行家のアルゴリズム**（banker's algorithm）”というが，これは前のパラグラフで述べた原則に基づいている［Dijkstra, 1968a］．新しい状態が安全であるための必要十分条件は，すべてのプロセスが完了できるということである．銀行家のアルゴリズムがチェックしているのは，この条件である．

そのアルゴリズム（**図 2-12** 参照）は，銀行が安全に貸し出しできるかどうかを決定する手続きに似ている．各プロセスは，$1 \leq i \leq N$ なる整数 i を持って識別する．プロセス i には最大要求（maxdemand [i]），現在割り当てられている個数（allocated [i]），残りの要求（claim [i]），およびフラッグ（maynot-finish [i]）が与えられている．システムは，全域変数として totalunits を持

```
unusedunits := totalunits ;
for i := 1 step 1 until N do
    begin    unusedunits := unusedunits − allocated [i]
             maynotfinish[i] := true ;
             claim[i] := maxdemand[i] − allocated[i]
    end ;
flag := true ;
do while (flag) ;
    flag := false ;
    for i := 1 step 1 until N do
       begin if maynotfinish[i] and claim[i] ≤ unusedunits
             then begin maynotfinish[i] := false ;
                        unusedunits := unusedunits + allocated[i] ;
                        flag := true
                  end
       end
    end
end ;
if unusedunits = totalunits then system is safe
                            else system is unsafe
```

図 2-12 銀行家のアルゴリズム

ち，システム内に存在する個数の合計を示している．初期状態では，どのプロセスも終了できるかどうかわらない（すべての i に対し，maynotfinish $[i]=$ true である）．未使用として残っている個数で，要求が満足されるときは，いつでもそのプロセスは終りまで実行され，その個数が解放されるものと仮定する．すべての個数がいつかは解放されるならば，すべてのプロセスが完了しシステムは安全である．もしも，システムが安全でなければ，その要求は認められない．

2.7.2 注 意 事 項

銀行家のアルゴリズムは簡単ではあるが，その実現には非常に費用がかかる．なぜならば，資源に対する新しい要求があるたびごとに，これを呼び出さなければならないからである．もっと簡単な方法は，各プロセスがその要求の最大数を一度にまとめて要求することである．その割り当てをふやすためには，プロセスは最初に現在持っているものを解放しなければならない．そのような方式を用いると，実際に必要となるときよりもずっと前に資源を要求するプロセスもあり得るが，システムは常に安全であるので，その実現は銀行家のアルゴリズムに比べて，ずっと簡単である．この方式はオペレーティング・システムでよく用いられる資源割り当ての方法である．

デッドロックは，重要でかつ興味のある問題である．これはオペレーティング・システムの問題の中では，形式化し易いものである．したがって，この分野に関してはたくさんの研究がなされている（たとえば，Coffman ほか [1971]，Habermann [1969]，Holt [1972]，Hebalkar [1970] など）．特に，Holt によるグラフ理論を用いたモデルを補遺 II に示してある．しかし，デッドロックに対する精巧な解析的扱いにもかかわらず，簡単な方式がやはりよく使われている．現場で動いているシステムにおいては，デッドロックの防止のためのオーバーヘッドは，一般にときどき生ずるシステムの故障に比べて非常に高価である．

演 習 問 題

2.1 ILLIAC IV の構成で代表されるような表面にあらわれた並列性と，IBM 370/195 の構成で代表されるような裏にかくれた並列性の違いは何か．ど

ちらの構成の方が機械の性能に関して大きな可能性を持っているか．どちらの機械の方がプログラムし易いか．それは何故か．

2.2 Dekker のアルゴリズムを実現している**図2-4**のプログラムを考えよう．プログラムの内部状態を定義し，また，計算過程におけるアルゴリズムの進行を記述する状態間の遷移関数を定義せよ．

2.3 **半順序**（partially ordered）のついた8個のタスクの集合 $\{A, B, C, D, E, F, G, H\}$ を考えよう．タスクAは，タスクC，B，E，より先に実行しなければならない．タスクEとDは，タスクFより先に実行しなければならない．タスクCとBは，タスクDより先に実行しなければならない．タスクFは，タスクGとHより先に実行しなければならない．

(a) タスク間のこのような順序を parbegin および parend 命令を用い，可能なところでは並列性を用いるようにして，プログラムとして記述せよ．

(b) タスクEが，タスクCより先に実行しなければならないという条件を，もう一つ加えるとしよう．parbegin および parend を用いて，やはり“最大の並列性”の記述が可能であるか．

(c) 腕木システムの命令が使用できるとしたら，(a)および(b)に対する答はどのように変えられるか．

2.4 第2.2節に示したプロセスの定義を考えよう．2つの $n \times n$ の行列の積を計算するような変数の集合と作用の集合とを定義せよ．使用可能な並列性を最大限に利用することができるか．

2.5 単一の危険部分を持ったn個のプロセスに対して，記憶の相互ロックを用いて相互排除を実現せよ．

2.6 生産者－消費者の関係を，記憶の相互ロックを用いてどのように実現するのか，概略を述べよ．相互排除を行なうためには Dekker のアルゴリズムをマクロとして用いよ．

2.7 **眠っている床屋の問題** [Dijkstra, 1968a].

床屋の店には，待ち合い室Wと理髪用椅子Bのある部屋とがある．Bから

WへおよびWから外部へ通ずるための引き戸Dがある．床屋はWを見て誰もいなければ，眠りにつく．お客が店に入ってきて，床屋が寝ているのを見つけたときは，彼を起さなければならない．待ち合い室には，有限の数n個の椅子しかない．床屋とお客とをプロセスとしてプログラムせよ．同期をとるために，メッセージの通信を用いよ．床屋が2人いるならば，どうなるか．床屋の店の動作を補遺 Ⅱ.2 節で概説する**ペトリ・ネット**(Petri net) を用いてあらわせ．

2.8 5個の椅子と5人分のセットの用意がしてある食堂のテーブルを考えよう．各セットの間には，フォークが置いてある．お客がスパゲッティを食べるためには，自分のセットの両側にあるフォークを両方使わなければならない．5人のお客が食卓に座り，スパゲッティを食べることと政治について議論することを，交互に行なっているものと仮定しよう（政治の議論に際しては，フォークは必要ないものとする）．通信のためには腕木を用いて，お客をプロセスの集合と考えてプログラムせよ．お客の中では，デッドロックが生じないことを確かめよ [Dijkstra, 1971].

2.9 CPU で実行される主プロセスM，リーダ・チャネルで実行されるリーダ・プロセスR，プリンタ・チャネルで実行されるプリンタ・プロセスPから成る非常に簡単なオペレーティング・システムを考えてみよう．Rがデータを読んでそれをMに渡し，Mはそれを変形してPに渡し出力するように，3個のプロセスのプログラムを作成せよ．このとき，次の2つの場合について作成せよ：

(a) 大きさがKであるようなバッファのプールが，2個ある場合．

(b) 大きさがKであるようなバッファのプールが，ただ一つある場合．

2.10 システムにおいては，たとえば，ファイルのようなかなり長いメッセージを，あるプロセスから他のプロセスへ転送したいことがあるであろう．標準の郵便受けシステムでは，記憶の中にメッセージを写さなければならないので，明らかに能率が悪い．短いメッセージを転送できるような能力を持っているときに，長いメッセージを転送する方法について概略を述べ

よ．

2.11 プロセスがメッセージの転送によって通信をしているようなオペレーティング・システムを考えよう．プロセスAがメッセージを転送するときには，返事を期待している．しかしながら，ときには返事をするプロセスの中で，何かうまくいかなくなり返事が絶対にこないようなことがある．Aが RECEIVE 命令を用いて自分は眠ってしまうと，それは永久に起されない．この問題を避けるために，Aが“目覚まし時計”をセットし，あらかじめ決められた時間以内に返事が来なかったならば，Aを起こして貰う．ハードウェアのタイマを用いて，目覚まし信号を要求したプロセスに送り出す特別のプロセスを設計せよ．

2.12 プロセスとプロセスの相互通信の機構とは，オペレーティング・システムにおいて，かなりのオーバーヘッドを作り出す可能性がある．非常に簡単でかつ効率のよいプロセス・ディスクリプタとプロセスの相互通信機能とを設計せよ．一般性と柔軟性とを，必要なところで犠牲にせよ．

2.13 一意的な整数で識別されるプロセスの集合を考えよう．各プロセスは制限のついた領域を持っているが，そこには制限領域内を実行中のプロセスが持っている整数値の総和が3で割り切れるとき，入ることができる．腕木システムを用いて，制限領域の実現方法の概略を述べよ．また，Hoare のモニタを用いた実現方法について概略を述べよ．2つの解を比較せよ．その実現方法では，プロセスが制限領域に対し永久にロックされてしまうことがないだろうか．

2.14 メッセージの転送を用いているシステムにおいては，メッセージのために記憶領域が必要となる．一つの方法は，送り主が自分の作業領域から領域を出すやり方である．もう一つの方法は，システムがこれらの目的のために管理しているバッファのプールを利用する方法である．2つの方法のそれぞれの利点を議論せよ．

2.15 モニタを用いて，2つのプロセスにおける生産者－消費者の関係を実現せよ．

2.16 計算機のネットワークを考えよう．ネットワークの異った節点で実行されているプロセスの間で使用できる通信機構について概略を述べよ．

2.17 デッドロックのモデルおよびデットロックの防止と検出のアルゴリズムについて調べよ．

2.18 補遺Ⅱ.2節に示してある例を，似たような並列処理のモデルについて調べよ．これらのモデルから得られるいくつかの理論的結果について概略を述べよ．

2.19 ペトリ・ネットとそれに関する理論的結果について，小論文を作成せよ．

2.20 ファイルのような危険資源を使用する2つのクラスのプロセス "reader" と "writer" があるものと仮定せよ．writer は資源の使用を排他的に行なう．writer がそれを使用していないときは，任意個の reader が，その資源を使用してよい．現在資源を使用している writer がいるときだけ，reader は使用を拒否されるものとする．同期のためには腕木システムを用いて，reader と writer とを並列プロセスとしてプログラムを作成せよ [Courtois ほか，1971]．

第 3 章

処理装置の割り当て

3.1 はじめに

処理装置は割り当てられる資源としては，システムの中で最も重要なものであると伝統的に考えられてきた．記憶や入出力装置の重要性がましてきたのにもかかわらず，処理装置の割り当てはやはり主要な働きをなしている．プロセスは，処理装置がなければその計算を進めることができないことは言う迄もない．今日の高速の機械を使えば，1 秒間に数百万回の命令の実行がなされる．処理装置にほんの少しの無駄時間があっても，失われる計算能力は非常に大きい．したがって，大抵のシステム設計者は処理装置を能率よく使用するために多くの関心を払うのである．

本章では，一つ以上のモジュールから成る計算の一つの単位として**ジョブ** (job) を定義する．各モジュールは機械語のプログラムおよび（または）いくつかのデータから成る．ユーザのジョブを効率良くスケジュールすることは，システムの仕事である．

本章では，まず処理装置の割り当て方式に関して，複雑さの順にしたがって概括する [Rosin, 1969]．次に，より進んだ環境，たとえば多重プログラミングについて議論する．最後に，いろいろのクラスのスケジューリングの方式とその実現法を紹介する．

3.1.1 専用機械

計算機の初期の頃は，計算機は一時には一人のユーザに割り当てられていた．今日では，小型の機械はこうして操作することが多い．この状況は**ミニコンピ**

ュータ(minicomputer)が発展すると共に，特に，よく見られるようになってきた．ユーザはテープやディスク・パックを装填し，コンソールに座って自分のプログラムを実行する．システムは，一般に，簡単で低速であり考える時間もあるので，その操作はそんなに難しくはない．ユーザのプログラムは，標準入出力やユーティリティのルーチンのようなシステムの支援を利用することができる．しかし，ユーティリティ・システムはユーザがひき起す誤りからユーザを保護することはできない．プログラムはどんな入出力に対しても，自分自身で同期をとらなければならない．テープやディスクのラベルの正当性を調べたり，コンソールにメッセージを送ったり，オーバーフローのようなプログラムの異常状態についてユーザに警告を発することなどをしなければならない．このようなことを支援するオペレーティング・システムは，存在しないかまたはほとんど役に立たない．ユーザは，実際，自分自身に頼らなければならない．

専用機械の**処理量** (throughput) はきわめて小さい．プログラマがテープを装填したり，カードを積んでいる間に失われる時間は大きい．また，機械自体も入出力命令を処理している間は一般に処理装置はアイドル状態なので，無駄に過すことになる．もっと高速で能力の大きい機械の場合には，そのような処理方式は非常に高価なものになるので，絶対避けなければならない．したがって大きなシステムに対しては，もっと効率の良い処理方式が必要である．

3.1.2 一括化 (batching)：ジョブをファイル化すること

計算機の設備のコストが増加するにつれて，長いアイドル時間はだんだん堪え難くなる．アイドル時間をへらすために，バッチ・システムではたくさんのジョブを入力テープに集めておく．ジョブは小型の“衛星”計算機で読み込み，入力テープとして準備する．それから，主処理装置が入力テープから一回に一つづつ取り出してプログラムを実行し，出力を出力テープに書き出す．その間には，衛星計算機は次に処理すべき入力テープを作り，前に終った出力テープをプリントする．代表的なバッチ・システムは IBM 7090 の IBSYS であり，IBM 1401 を衛星計算機として用いている．

入力テープ上のジョブは，先来先処理に従い処理される．したがって，もしも計算機が入力テープの処理に1時間費すならば，ジョブの平均の**ターン・ア**

ラウンド・タイム（turnaround time）は，衛星計算機の速度にもよるが大体２～３時間となるであろう．この方式では短いジョブは特別扱いをしているわけではない．15分かかるジョブは，15秒かかるジョブと同じターン・アラウンド・タイムを持っている．したがって，処理の費用のほかにはプログラムの実行時間をへらそうとするような動機は生じてこない．

バッチ処理は単純な処理方式である．それは専用機械に比べてかなり処理量の向上が見られる．しかしながら，プログラマとプログラムとの間の相互作用が全然ないので，デバッギングはかなり難しくなるであろう．それゆえ，バッチ処理方式は周期的なスケジュールにしたがって処理される商業用システムでは最もうまく働く．

3.1.3　直接結合機械

直接結合システムにおいては，２個（またはそれ以上）の計算機が記憶間の通信を介して結合されている．**マスター**（master）と呼ぶ低速の機械は**スレーブ**（slave）と呼ぶ高速の機械に対するジョブのスケジューリングの責任を持っている．マスター計算機はジョブをカード・リーダから読み取り，ディスク上の待ち行列に置く．スレーブ計算機がジョブの実行を終えたときには，マスター計算機に対して次の仕事の要求を送る．スケジューリング，料金計算および入出力は，マスター計算機が権限を持つ．スレーブ計算機は各ユーザ・ジョブに対して，計算能力そのものを責任を持って提供している．それは生産的な仕事にすべてを打ち込むために，管理的な仕事は放棄しているのである．

ジョブのスケジューリングは非常に複雑になる可能性があり，また，そのシステムが提供しようとしている処理の性質に大きく左右される．スケジューリングの原理についての議論は，第3.３節にて行なう予定である．

3.2　多重プログラミング

多重プログラミング・システムにおいては，いくつかのジョブを，一時に一部分づつ実行している．一般に，ジョブは主記憶内にあり，主処理装置はそれらを各々少しづつ実行しているのである．多重プログラミングを用いる主なる理由は，命令処理と入出力操作との間の速度の違いを補償するためである．入

出力操作は命令実行の速度に比べて遅い．一つのジョブがチャネルに対して読み込み命令を出したときは，チャネルがその読み込みを完了するまで待たなければならない．多重プログラミング・システムにおいては，入出力操作の間，主処理装置をアイドル状態にしておかないで，処理装置を異ったユーザのジョブの実行のために切り替えることができる．現在，入出力を実行しているジョブは，その入出力が完了してから処理装置の支配権を獲得する．主記憶内には複数個のジョブが同時に存在しているので，それらがすべてチャネルの完了を待っている場合だけ，処理装置はアイドルになる．

ジョブは主記憶内にあり，実行が可能なとき**アクティブ**（active）であるという．アクティブなジョブの数は，制限したり変更したりすることができる．処理装置がほとんどアイドルにならないために，出来るだけ多くのアクティブなジョブを持っていることが望ましい．しかしながら，ジョブがアクティブになるためには主記憶が必要であるが，それは記憶空間を得るための深刻な争いをジョブ間に作り出す．したがって，記憶管理の方式では，効率的に記憶を詰めることができなければならないことは重要なことである．

システム内のアクティブなジョブの集合を**ジョブ・ミックス**（job mix）という．ジョブ・ミックスの選択は多重プログラミング・システムにおいては，重要な決定の一つである．一般に，ほとんどの時間は計算をし，入出力をほとんど行なわないような“**計算限界**（compute bound）”のアクティブ・ジョブが，少なくとも一つは存在することが望ましい．たとえ，他のすべてのジョブが入出力を行なっていても，計算限界のジョブは処理装置を働かせ続けることができると考えられる．コア記憶の中に同時に多数のジョブがいることが望ましいので，少量の記憶しか必要としないジョブは非常に有用となる．“小型のジョブ”はジョブの命令コードとデータとの一部分をコアに置いておくことにより，人工的に作り出すことができる．ジョブの残りの部分は，ドラムやディスクに置いておく．命令コードやデータが必要なときは，ジョブは一時的に中断し（すなわち，アクティブでないようにし），周辺装置から追加の情報を持ってくる．このような記憶管理の方式によれば，ジョブの一部分しかコア記憶の中にいないので，多くのジョブをアクティブにすることができる．

処理装置のアイドル時間は最小にしなければならないが，ほかにもいくつか

の目的がシステムにはある．周辺装置も，また，出来るだけ効率よく使わなければならない．システムはテープ駆動装置，ディスク駆動装置，カード・リーダ，ライン・プリンタを動かし続けるようなアクティブなジョブを選択しなければならない．もしも，すべてのアクティブなジョブがディスクを使用しどのジョブもテープを使用しないならば，システムは明らかに不釣り合いである．ジョブの資源の要求を事前に知ることは，明らかに非常に役に立つ．しかしながら，システムの**釣り合いをとること**（balancing），すなわちすべての資源を使用させ続けることは非常に難しい問題であり，最適解がほんの簡単な場合にしかわかっていない．

3.2.1 時 分 割

時分割（time-sharing）**システム**は，多くのユーザに対し，各々高速な処理を与えながら相互に作用を行なっている．多重プログラミング・システムはそのような処理を実現する一般的な形態である．多くのユーザの要求を高速に処理するためには，多くのジョブを主記憶に同時に格納しておいて初めて可能である．このシステムの応答の速度により，人間－機械系の通信に大きな改善が可能となる．時分割システムは専用機械におけるユーザとの密接なつながりと，バッチ・システムの効率とを結びつけている．ユーザへの同様な環境は，プロセス制御においては**実時間**（real-time）あるいは事務処理の応用においては**オンライン**（on-line）と呼んでいる．

3.2.2 時間を刻むこと

多重プログラミングの環境においては，アクティブなジョブの間で処理装置を共有しなければならない．共有を実現するためには，処理装置の時間をある間隔に分割し各ジョブは何回もその間隔を割り当ててもらう．各間隔のことを，**時間刻み**(time slice）または**刻み**（quantum）という．たとえば，5人のユーザのジョブがコア記憶にいるとき，各々には5秒ごとに1秒の時間刻みを割り当ててよい．しかし，時間刻みはすべてのジョブに対して，必ずしも同じ長さである必要はない．ジョブの優先度や処理装置の必要性などの要因が，一般に，ジョブの時間刻みの大きさやそれを割り当てる頻度などを左右する．

時間を刻むことは，各ユーザが連続的な処理を期待しているような時分割システムにおいては，特に重要である．短い時間刻みを用いると，各ユーザは機械が自分だけを実行しているかのような錯覚をおぼえる．システムは，1秒ごとに0.1秒の時間刻みをユーザに与えるかもしれない．ユーザは実際のハードウェアの十分の一で動いているような機械で，自分のジョブが実行されているように見えるだけである．

複数個のジョブに対して，時間を刻むアルゴリズムはたくさんある．この章の残りの部分では，多重プログラミングのスケジューリングでよく使われるものをいくつか紹介しよう．

3.3 多重プログラミングのスケジューリングの方法

多重プログラミング・システムでは，多数のアクティブなジョブが処理装置の時間を競い合う．システムの中で，各ユーザのジョブはプロセスとして実行すると仮定しよう．ユーザ・プロセスのほかに，処理装置の使用を共有するシステムのプロセスが存在する．処理装置の支配権を得るために待っているすべてのアクティブ・プロセスは，**実行可能待ち行列**(ready queue)の中に置いてある．停止しているプロセス（たとえば，入出力完了や腕木が正になるのを待っているプロセス）は，それぞれの**停止待ち行列**（blocked queue）に置いてある．

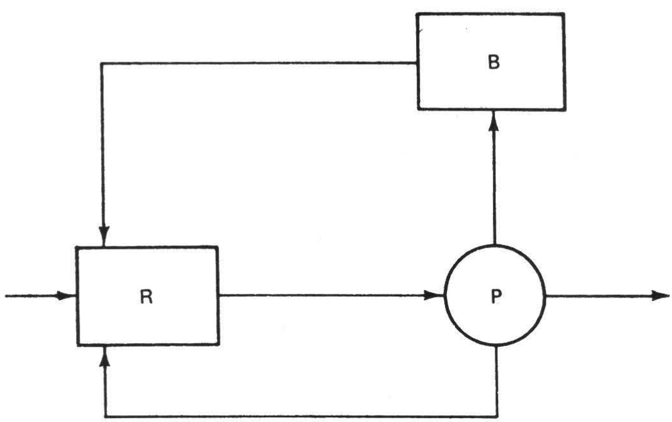


図 3-1 処理装置のスケジューリングの図式表現（R，実行可能待ち行列；B，停止待ち行列；P，処理装置）

図 3-1 はこれらの待ち行列の図式的表現である．R は一つ以上の実行可能待ち行列であり，P は処理装置である．これらの待ち行列に対応して，プロセスは次のうちのどれか一つの状態になっている．すなわち，**実行可能** (ready)，**停止** (blocked) および**実行中** (running) である．実行可能待ち行列の中から取り出したプロセスは，処理装置に対する時間刻みを割り当ててもらう．それは自分の時間刻みがつきてしまうか，（たとえば，入出力操作によって）停止するか，終了するか，あるいは処理装置の支配権を与えられた他のプロセスによってそれを取り上げられるまで実行を続けている．いま述べた4つの理由のどれかによって，その時間刻みを完了すると，プロセスは実行可能待ち行列へ戻されるか，停止待ち行列に置かれるか，あるいはシステムから取り除かれるか，どれかが行なわれる．停止待ち行列に置かれたプロセスはその停止がとかれたとき，実行可能待ち行列に移される．この待ち行列のシステムに対する**スケジューリングの方針** (scheduling discipline) は，実行可能プロセスを R から選び出し，それに時間刻みを割り当てる方法を決めることである．

実行可能待ち行列に於けるプロセスには，プロセスの重要度を示すために，それぞれ**優先度** (priority) が与えられている．優先度はプロセスの静的および動的な性質に基づいて計算される．静的性質とは，そのプロセスの大きさ，推定実行時間，入出力動作の推定量などをさす．プロセスがユーザのジョブであるならば，ユーザが選んだ料金の率，すなわち一般に**外部優先度** (external priority) と呼ぶジョブのデックの最初に書かれている優先度の数も，1つの静的な要素と見なしてよいだろう．優先度は，また，システム内における動的なふるまいによっても左右される．優先度の計算に際しては，プロセスの現在の資源の所有量，待ち時間の総和，処理時間の量，入出力の相互作用の量，システム内に存在した総時間などが考慮される．これらの性質は時間と共に変化しているので，プロセスが存在している間に優先度は何回も計算してよい．

大抵のシステムにおいては，実行可能待ち行列に対する優先度の計算に際しては，外部優先度は大した役目をするわけではない．外部優先度は最初にジョブがアクティブな状態になるときに主として使用されるのであり，また，いったんその実行が完了したときには，そのプリントを急いで出力するときに役に立つのである．大抵のジョブの実行時間はコアにロードされるのを待っている

時間やプリントを待っている時間に比べて非常に短いので，実行可能待ち行列の優先度を外部優先度によって大きくしてもジョブのターン・アラウンド・タイムにはほとんど影響せず，ジョブ・ミックスの釣り合いという点に関しては悪い影響しか与えないだろう．

多くのシステム・プロセスはユーザ・ジョブのようにして実行される．料金計算，ルーチンの管理および保守はこの部類に属する．しかし，いくつかのシステム・プロセスは特別な権利が必要となる．たとえば，周辺機器の管理システムは処理装置を使用できる状態になったときは，それを待たしてはならない．それが処理装置を与えられるまでは，それに付随しているチャネルは，恐らく，**開始命令**（start command）が出されるのを待って，アイドル状態になったままであろう．周辺装置のアイドル時間を避けるためには，ある種のプロセスは特権的地位にあり，ユーザ・プロセスよりもずっと簡単に処理装置を得ることができなければならない．特定のスケジューリング・アルゴリズムを議論するときには，この種のプロセスについて特に言及はしないけれども，論理的にはすべてのスケジューリングの方針は特権的プロセスも扱えるようになっていなければならない．たとえば，**最高優先度先処理**（highest-priority-first）のアルゴリズムにおいては，特権的プロセスには，一般に，無限に大きな優先度を与えておく．

実行可能待ち行列のスケジューリングの際には，システムは，資源の使用効率のよいこと，処理装置のアイドル時間の低いこと，大きな処理量，ユーザ・ジョブに対する妥当な待ち時間などのいくつかの性能指標を考えなくてはならない．さらにどのジョブもシステム内で永久に待たされることがあってはならないという条件も課せられる．ときには，目標が矛盾することもある．たとえば，平均応答時間を改善するためには，いくつかの長いジョブは実行開始前には長時間待たされるかもしれない．スケジューリングの規則を評価する一つの方法は損失関数を用いることである．**損失関数**(loss function)はジョブを完了するための値を遅れ時間の関数として表現している．損失関数に対する重要なパラメータは，待ち時間，システム内に滞在する総時間，**期限**(deadline)などである．システムは平均損失を最小にするかまたはすべてのジョブに対する最大の損失をある制限内に押さえるか，どちらかの方法でスケジュールをするよ

うに努める．しかしながら，損失を最小にすることは，システムの使用効率を上げることとは一致しないかもしれない．たとえば，平均待ち時間を最小化すると長いジョブの実行が待たされる．もしも，長いジョブが共有不可能な資源を専有しているときは，その長いジョブを早めに実行した方がその資源をずっと効率良く使用できる．長いジョブを実行すべきかどうかを選択するのは，どの性能指標がより重要であると考えるかによって左右される．一般に，実際のシステムにおけるスケジューリングの規則では，多くの目標に関して妥当な性能を発揮するようにしている．単独の目標を最適化することはないけれども，全体の性能は良いものが得られる．

この節では，3つのスケジューリングの規則を議論する．すなわち，最高優先度先処理，**ラウンド・ロビン**（round robin），フィードバック待ち行列の3つである［Coffman および Kleinlock, 1968］．先来先処理（FCFS）については，比較のためにときどき言及する．この3つの基本的な法則は大部分の処理装置のスケジューリングの問題を含み，また，他の資源割り当て問題の多くの場合に一般化されている．

3.3.1　最高優先度先処理

実行可能待ち行列をスケジュールする一つの単純な方法は，最高の優先度を持つプロセスに処理装置を割り当てる方式である．途中で先取り（preempt）されることがない場合は，最高優先度のプロセスは終了するか，自分自身を停止するかどちらかが起るまで実行を続ける．現在実行中のプロセスよりも優先度の高いプロセスが到着したときは，実行中のプロセスが処理装置を放棄するまで待たなければならない．先取りが許されるときは，優先度の高いものが到着すると，実行中のプロセスに割り込んで自分が支配権を取ってしまう．先取りされたプロセスは実行可能待ち行列に戻される．**最高優先度先処理**（highest-priority-first；略して **HPF**）方式では，優先度の要因，先取りの使用の有無，実行可能待ち行列の構成などは実施する際に自由に取り決めてよい．

処理装置が空いているときは，最高優先度のジョブが実行可能待ち行列から取り出されなければならない．待ち行列が分類されているならば，先頭の要素を取り出すのは簡単なことである．しかし，待ち行列へ挿入するときには時間

がかかる．すなわち，待ち行列の先頭と最後尾の要素へのポインタを持った**二重結合リスト**（doubly linked list）で構成されていると仮定したとき，到着するたびに待ち行列の半分を走査しなければならない．待ち行列が分類されていないときは，新しいプロセスが必要なときは，いつでも捜さなければならない．しかし，挿入は簡単である．両方を妥協させた方式も，また可能である．たとえば，待ち行列をT時間単位毎に分類してもよい．プロセスが到着したときは，それを待ち行列の先頭につけ加える．しかし，分類したリストの部分の先頭要素に対するポインタPは保持しておく．最高優先度のプロセスを選ぶために，走査はPから始まり待ち行列の先頭へ続けてゆく．走査しなければならない要素の数は，分類しない場合に比べて少なく，また，分類に要する時間は厳密に分類する場合に比べて少なくなる．しかし，簡単のために，一般にはシステムは分類した実行可能待ち行列を使用している．

HPF方式を用いたときに決めなければならないことは，どの性質を優先度を決める基本にすべきかということである．HPFで一番頻繁に言及される方式は，**最短ジョブ先処理**（shortest-job-first；略して **SJF**）方式である．すなわち，優先度は処理時間に反比例して決められる．事前にプロセスの実行時間を正確に知ることは難しいので，推定値が優先度決定のために通常用いられる．少ない実行時間を推定されたプロセスは，あきらかに有利である．ユーザは自分の優先度を上げるために，推定値についてうそをつくかもしれない．このようなふるまいは推定時間を越えて実行されたようなユーザを罰することにより，押さえることができる．たとえば，推定値を越えるようなプロセスには料金をふやすことができる．

SJF 方式は，先取りを使用してもしなくても，実施することができる．先取りのない場合には，プロセスはそれが終了するかまたは停止するまで，実行をしている．先取りのある場合には，実行中のプロセスに比べて少ない推定実行時間を持ったプロセスが到着したときには，実行中のプロセスは先取りされ，新しいプロセスが取ってかわる．先取り方式は，短いプロセスにとっては先取りがない方式に比べて有利である．先取りがある場合，長いプロセスにとっては処理装置を持ち続けることは困難になる．

先取りは，短いプロセスにとってよい処理を与えてくれるけれども，それに

伴う費用がかかる．先取りを行なうために必要な時間は，実行可能待ち行列の中のすべてのプロセスに遅れをもたらす．したがって，実行中のプロセスがほとんど終了に近いときならば，それを先取りする意味はない．

典型的な実行時間の分布を見ると，“短い”プロセスが圧倒的に多い．このような環境の中では，SJF 規則は短いジョブを非常に優利に扱うので，まったくうまく働く．しかし，長いジョブはよい処理をして貰うことができない．システムが非常に混んでいるときは，長いプロセスは先取りが使用されていれば特にひどい目に会う．しかしながら，システムが飽和状態に達したときは，どのスケジューリングの方法も，特によく働くということはない．

実行時間に関する事前の情報は，一般に，非常に不正確であるので，多くのスケジューリングの規則では，プロセスがシステムにいる間に得た情報に基づいて基本的な決定をしようとしている．ある規則のクラスでは，線形的に増加する優先度を使用している ［Kleinlock, 1970］．各プロセスはシステムに入ってきたときに，一つの優先度を割り当てられる．その優先度は実行可能待ち行列で待っているときには比率 a で，また，実行されているときには比率 b で増加してゆく．a や b をどのように選ぶかによって，異ったスケジューリングの規則が得られる．たとえば，$0<a\leq b$ ならば先来先処理（FCFS）の規則となる．また，$0>b\geq a$ ならば**後来先処理** (last-come-first-served；略して **LCFS**) の規則となる．このクラスに関する最初の論文の中で，Kleinlock はパラメータ a と b を変えることによって得られるすべての可能な規則を述べている．次節では，このクラスから得られる一つのアルゴリズムについて議論する．

線形増加優先度方式は多くの場合に展開できるけれども，それらのうちほとんどのものは解析されていない．たとえば，比率と a と b はプロセスの外部優先度，すなわちどの位のお金を支払う気があるのかということに基づいて決めることができる．また，非線形の関数を用いることもできる．優先度を時間と共に減少させてもよい．最大時間に到達したときには，優先度をある高い値に移す．こうすれば処理をして貰うためにはどんなプロセスも非常に長く待たせることはないという条件の下で，短いプロセスを有利に扱うことができる．同様の規則を SJF に追加することができる．すなわち，長いプロセスはしばらくシステムの中に居て優先権を上げて貰ってから開始する．システムが扱うプ

ロセスのタイプの微妙な意味が，選択する優先度方式のタイプを大きく左右する．

スケジューリングの規則の一つの重要な目標は，周辺機器を使用中の状態にさせておくことである．この目的のために，プロセスは入出力動作の多い間は，高い優先度を与えてよい．プロセスは入出力命令を出した後は，一般には停止待ち行列に入っているので，そのような動作を見張るのは容易に行なえる．特に，プロセスは最後の入出力命令以後の時間に反比例した優先度を持ってもよい．プログラムは，その入出力を集中的に行なう傾向があるという実験結果がある．入出力動作に基づいた HPF 方式を用いれば，ほとんどの入出力を実行しているプロセスが一番処理装置を得る機会が多いので，周辺機器はほとんどの時間，使用中の状態に保っておくことができるであろう．

あきらかに優先度規則の数は理論的には無限にある．多くのプロセスによる処理を与えるのには，HPF は効率的ではあるけれども，それは低い優先度のプロセスを長く待たせてしまうという側面的効果を持っている．いくつかのシステム構成においては，このような状況は受けいれられない．たとえば，時分割システムにおいては，ユーザは1時間毎に12分間の処理をするよりも，5秒間毎に1秒間の処理をして貰った方がずっとよい．HPF はこのような処理を保証することはできない．したがって，二番目の重要なスケジューリングのクラス，すなわちラウンド・ロビンを思いつくのである．

3.3.2　ラウンド・ロビン

単純な**ラウンド・ロビン**（round robin；略して **RR**）のスケジューリング規則では，静的および動的な優先度の情報は何も用いない．実行可能待ち行列の先頭のプロセスは q 秒間の時間刻みを与えられ，それが自分で停止しない限りは，実行可能待ち行列の最後尾に戻される．実行可能待ち行列の中に K 個のプロセスがあるときには，各プロセスは処理装置の時間の Kq 秒間に q 秒間を得ることになる．したがって，各プロセスは物理的な処理装置の速度の $1/K$ の速度で実行しているように思える．それゆえ，実行可能待ち行列の大きさはプロセスがどれだけ速く進行するかを決める重要なパラメータである．

刻みの大きさ q は，短い周期の間に，どれだけ平等に処理装置が分布される

かということを決める．qが無限大であるならば，RRはFCFSになる．プロセスはいつでも実行可能待ち行列に入ったり出たりしているので，大きな（有限の）刻みの大きさを用いると，ランダムにいくつかのプロセスを有利に扱うかもしれない．たとえば，一つのプロセスがその刻みを実行し終ったときにちょうど4つのプロセスが到着したならば，そのプロセスの次の刻みは$4q$遅れて割り当てられる．一方，あるプロセスは自分の前にいる多くのプロセスが停止しているので，次の回に対する待ち時間を減らすことができるような幸運にめぐりあうかもしれない．qが小さくなれば，一つの刻みの間に起る到着や停止の確率が小さくなるので，そのようなランダムな効果は減少する．

qが減少すると，小さいプロセスに対する処理は改善される．qが非常に小さいときは，すべての実行可能なプロセスは平等な処理をされ，待ち時間は必要な処理の時間に比例する．しかしながら，qは必要以上に小さく選んでしまう可能性もある．“妥当な大きさ”の刻み幅であれば，一つのプロセスから他のプロセスへ実行を切り替えるのに必要な時間は無視できる．qがプロセスの切り替え時間と同じオーダの大きさであるときは，切り替えによる遅れは無視できない．実際，qが小さ過ぎると，システムはプロセスの実行よりもそれらの切り替えのために，多くの時間を費してしまう．したがって，RRに於ける時間刻みの大きさは，容認できないような切り替えのオーバーヘッドが生じないような中間的な値を選ばなければならない．

RRは多くのユーザを処理する時分割システムにおいて，しばしば使用される．そのようなシステムは，一般に各プロセスに対して，妥当な応答時間を保証しなければならない．すなわち，刻みの大きさにかかわらず，ユーザは数秒間に少なくとも一回は処理されることを期待する．qが一定で待ち行列にはK個のプロセスがあるときには，応答時間は，大体，Kqになる．大きなKに対しては，これは大き過ぎるかもしれない．**サイクル型**（cycle-oriented）の*RR*規則を用いれば，応答時間が保証される．この方式では最小限容認できる応答時間になるように，実行可能待ち行列に対するサイクル時間が選ばれる．実行可能待ち行列における各回の開始時において，サイクルの大きさを待ち行列内のプロセスの数で割ったものを，刻みの大きさにする．1サイクル中に到着したプロセスは実行可能待ち行列に加えられる前に，サイクルの完了するまでとめておく．

待ち行列が非常に長いときには，q が極めて小さくなるので，切り替えによる損失が無視できない．この問題は最小の時間刻みを決めておくことにより，解決することができる．非常に混んでいるときは，応答時間は悪くはなるけれども，少なくとも切り替えによるオーバーヘッドは，低いレベルで押さえられる．

RR 方式に対する他のいくつかの変形方式が考えられる．**下駄ばき**（biased）RR は1つの方式であるが，それは各プロセスに対しその外部優先度に基づいた刻みを割り当てる．各プロセスはそれぞれの刻み幅を持つので，その優先度に比例した速度で実行が進められる．同様の方式が，他の優先度の要因，たとえば資源割り当て，入出力動作，推定処理時間などに基づいて考案できる．もう1つの RR 方式は，Kleinlock の線形増加優先度方式（第3.3.1節参照）から得られる．パラメータ a と b とが，$0 \leq b < a$ を満すとき**自己中心的**（selfish）RR と呼ばれるものになる．システムに入ってくると，プロセスは実行中のプロセスの持つ優先度の大きさになるまで待たされる．それから他の実行プロセスとともに，RR の中で実行される．実行しているプロセスは，$b < a$ なる比率でその優先度を増加してゆくので，待っているプロセスは，いつかはつかまえられる．自己中心的 RR および RR では，短いプロセスはその全実行時間中の大部分を一つの刻みで得てしまうので，FCFS に比べて短いプロセスを有利に扱っている．また，$b=0$ ならば，自己中心的 RR は単純 RR になってしまう．

一つ以上の実行可能待ち行列を用いることにより，システムは実行時間の分布を利用することができる．たとえば，プロセスが3個の時間刻み以内に完了しないときは，それが12個の時間刻みを得る以前には完了しない確率が85%であることがわかっていると仮定しよう．**フォアグラウンド**（foreground）および**バックグラウンド**（background）の2つの待ち行列を使用することによって，この実行時間の性質が利用できる．各プロセスはフォアグラウンドの待ち行列内で，3刻みだけ進める．3刻みの処理以後は，バックグラウンドの待ち行列に落とされる．処理装置はその大部分をフォアグラウンドのプロセスの実行のために用い，短いプロセスに対して，非常によい処理をしてくれる．サイクル型の方式では，フォアグラウンド待ち行列のプロセスには一定の刻みを与える．すべてのフォアグラウンドのプロセスを実行した後でも，サイクル内に

時間が残っていれば，その時間はバックグラウンドのプロセスの実行のために用いる．このような2つの待ち行列を持ったシステムは，多重レベルのフィードバック待ち行列に一般化することができる．

3.3.3 フィードバック待ち行列

基本的な**フィードバック**（feedback；略して**FB**）方式は，それぞれ FCFS で処理される n 個の実行可能待ち行列を用いる．システムに到着したプロセスは，一番目の待ち行列に入る．1刻みを受け取った後では，2番目の待ち行列に落とす．次の刻みを受け取った後で，さらに次の高い待ち行列に入れる．処理装置は，空でない一番数の小さい待ち行列を処理するようにスケジュールする．FB 方式では，新しく到着したものは高い優先度を持つことになり，システムにおいて実行されているプロセスで最も数の小さいものと同じだけの時間刻みの間，実行される．FB 方式は実行時間に関する事前の情報なしに，短いプロセスに対して，RR よりもよい処理を提供している．しかし，多数の待ち行列を保持してゆくためには，特に刻み幅が小さい場合には，多くのオーバーヘッドが必要になってきてしまう．

必要な待ち行列の数をへらす一つの方法は，サイクル型のフォアグラウンド－バックグラウンドの場合を拡張することである．各刻みの後で待ち行列の内容を変更することはしないで，プロセスが次の優先度に落ちる前に，各 FB 待ち行列内を決められた回数だけ回ることを許す．FB 待ち行列内では，同じ処理装置割り当て規則を適用して RR 待ち行列として処理する．異った大きさのプロセスの間で差別をつけることはうまくはないけれども，待ち行列の保持のためのオーバーヘッドは減少する．

ユーザはそのジョブを短いプロセスに分割することにより，FB 方式を上手に利用することができる．一つのジョブを単一な長いプロセスとして提供するのに比べて，全体の処理は驚くほど改善される．

FB と RR とが HPF に比べて良い点は，短い要求に対してよい処理を得るために，実行時間に関する事前の情報が必要ではないことである．一方，優先度がプロセスの資源の専有量の関数であるとき，資源の使用効率のためには，高い優先度を有利に扱うのがよい．優先度に関する情報を考えた方がよいときに

は，下駄ばきRRのような複合方式を用いれば，動的および静的データの両方の使用の釣り合いをとることができる.

3.4 多重レベル・スケジューリング

前節では，処理装置のスケジューリング方式についていくつかのものを紹介した．これらの方式では，ほとんどのものが待ち行列の処理，プロセスの切り替え，また優先度や刻みの計算などが必要であった．この節では，効率よく処理を行ないかつスケジューリングの決定が柔軟に行なえるような，スケジューリング規則の実現をする一般的な方法を述べる.

3.4.1 処理装置のスケジューラの実現

大抵のシステムでは，処理装置は割り込みを介して相互通信を行なう．主処理装置は**インターバル・タイマ**（interval timer）の期限がきたとき，チャネルが修了したとき，実行中のプログラムが入出力をしようと思ったときなどには割り込みに出合うであろう．割り込みが生ずると，主処理装置の制御はスーパーバイザやOSの核心に移る．スーパーバイザはいくつかの作用，たとえばチャネルの開始，プロセスの停止を解くこと，プロセスを生成することなどを行ない，それから実行可能待ち行列内の一つのプロセスに，処理装置の制御を戻す．割り込みは頻繁に生ずるから，実行可能待ち行列内のプロセスに処理装置を割り当てるたび毎に，非常に多くの命令を実行するのは能率が悪い．しかし，処理装置の現在の状態に関する情報を退避したり，優先度や刻み幅に関する幾つかの計算を実行することは重要である．したがって，一つの目標はこれらの計算をあまり何回も行なわないでも済むような，実行可能待ち行列の管理の方法を工夫することである.

この方法の鍵となる方針は，頻繁に生ずるような操作はあまり頻繁に生じない操作に比べて，時間のかからないようにしておくことである．この方針はオペレーティング・システムの設計ではよく使われるが，特にスケジューリング・アルゴリズムの場合にも適用でき，実行の頻度に基づいて操作をレベルに分けておく．ここで考えている**多重レベル・スケジューリング**（multilevel scheduling）の方法は，**ディスパッチャ**(dispatcher)，**短期スケジューラ**(short-

term scheduler) および**長期スケジューラ**(long-term scheduler)の3つのレベルから成るシステムである．ディスパッチャは，割り込みの処理が完了した後で呼び出される．これは実行可能待ち行列から次に実行すべきプロセスを選び出す．ディスパッチャは非常に頻繁に実行されるので，非常に短いものでなければならない．最良の場合には，実行可能待ち行列からその先頭のプロセスを取り出し，処理装置をそれに割り当てるだけでよい．短期スケジューラは実行可能待ち行列にプロセスを入れるときに呼び出す [Brinch Hansen, 1971]． このスケジューラはプロセスの状態を調べなければならないかもしれないが，優先度の修正はこのレベルではなるべく最少にしておかなければならない．プロセスは頻繁に，恐らく2～3ミリ秒毎に実行可能待ち行列に挿入される．したがって，プロセスの状態を複雑に修正することは，長期スケジューラに任さなければならない．長期に関する修正はめったに行なわれないが，もっと複雑なものとなってよい．資源の割り当て量のようなプロセスの状態に関する情報は，優先度を計算し直すために用いてもよい．長期スケジューラは2～3秒に一回しか呼び出されないので，その実行に数ミリ秒間かけることができる．

このような3つのレベルの構成は，スケジューリングのオーバーヘッドを分散させるという利点がある．呼び出される回数の予測値があらかじめわかっているときは，各レベルにおいて許される計算量の制限値が予測できる．実行可能なスケジューリングの規則は，どれでも各レベルにおける時間に関する制限を満足することができなければならない．

3.4.2　多重レベル・スケジューリングの例

第3.3.2節においては，RR方式を述べているときにプロセスの速度という概念について議論した．プロセスの速度は各々に割り当てられる刻み幅を調整することによって，変えられるものであることを示した．そこでプロセスの速度は，実行可能待ち行列の中を一廻りする時間で刻み幅を割ったものとして計算される．この方式はプロセスの速度に関して適当な管理をすることができるけれども，プロセスがどれだけ頻繁に制御を得ることができるかということは，何も言ってない．バッチ処理と時分割処理を両方行なうようなシステムを考えてみよう．バッチ処理は，処理装置の時間の75%を必要とするが，いつそれ

が得られるかということは問題ではない．時分割処理は処理装置の時間の25%だけしか必要としていない．しかし，1秒間に一回は，制御権を得なければならない．さらに，いくつかのプロセスは両方のサブシステムの下で動いており，これらのプロセスは，どれもその親のタイミングの制限条件を満足しなければならない．すなわち，時分割処理のプロセスは，どれも1秒間以上処理装置からしめ出されることがあってはならない．サイクル型の方式では，サイクル時間という上限を保証できる．しかしながら，この上限はすべてのプロセスにとって，同じ値でなければならない．もしも“実時間”処理が行なわれているときには，実時間処理のプロセスは10ミリ秒毎に20マイクロ秒を得なければならない．したがって，RR 待ち行列に対するサイクル時間は，10ミリ秒でなければならない．これはプロセスの切り替えの費用が実現できないほど高くなるので，あきらかに実施不可能である．この問題に対してプロジェクトSUE で J. J. Horning が提案した解は，多重レベルのスケジューラを用いることであった．

提案されたスケジューリング・システムにおいては，各プロセス $P(i)$ には刻みの大きさ $q(i)$ と**時間枠** (time frame) $f(i)$ とが与えられている．時間枠はどれだけ頻繁に，そのプロセスが刻みを受け取るかを指定している．プロセス $P(i)$ は実行可能待ち行列に入ってきたときに，$f(i)$ 秒以内に $q(i)$ 秒の処理装置の時間を受けとるはずである．プロセスの速度は $s(i)=q(i)/f(i)$ と定義されるが，それは停止していないときに受け取ることのできる処理装置の割合を示している．たとえば単純なサイクル型の RR 方式では，$f(i)=f$ は待ち行列におけるサイクル時間であり，また全体でK個のプロセスがあるとき，各プロセスは刻み f/K を受け取る．

実行可能待ち行列に入った後では，プロセス $P(i)$ は**期限** (deadline) $T(i)$ を持つ．すなわち，その刻みを受け取るまでの時間である．もしも，$P(i)$ が時間 $t(i)$ に待ち行列に入ってきたときは，$T(i)=t(i)+f(i)$ となる．次に，実行可能待ち行列は期限の大きさによって分類される．待ち行列の先頭のプロセスは，その刻みだけ実行されるか，または自分で停止するまで実行される．刻みが完了したときに停止状態でなければ，プロセスは実行可能待ち行列に戻され，新しい期限が計算される．

この3つのレベルのスケジューラに必要な命令コードの量は，各レベルにおける時間の制限と一貫性が保たれていなければならない．ディスパッチャは待ち行列の先頭の要素を取り出すだけでよい．短期スケジューラは期限を再計算し，そのプロセスを待ち行列に入れる．長期スケジューラは各プロセスに対し，$f(i), q(i)$ および $s(i)$ を調整できる．さらに，この単純な3つのレベルのシステムを調整することにより，これまでに述べてきたいろいろのスケジューリング・アルゴリズムを近似させることができる．たとえば，RR 規則はすべてのプロセスに対し，$q(i)=q, f(i)=f$ とすることによって実現できる．$q(i)$ が各プロセスの処理装置の必要量を越えているならば，このアルゴリズムは FCFS になってしまう．他の方式も同様にして得ることができる．

SUE アルゴリズムは洗練されてはいるけれども，いくつかの重大な限界がある．これはすべてのプロセスがコア記憶にあるものと仮定して設計された．この仮定は多重プログラミングを用いたバッチ処理では成り立つかもしれないけれども，時分割システムにおいては，一般には成り立たない．大抵の時分割システムでは，いくつかのプロセスに対するコアの空間を用意しているだけである．ユーザがその端末を介して通信をすると，プロセスは周辺装置からコアに持って来られ，他のプロセスはコアからディスクへ掃き出される．問題は，他のプロセスを持ってこれるようにするために，どうやって実行中のプロセスを停止させるかということである．あきらかに，他のどれかのプロセスを停止しなければならない．しかし，それは長期スケジューラによって開始しなければならないのだから，処理装置を得るのが遅れるかもしれない．すべてのプロセスがコアの中にいる限りは，事はうまく運ぶ．プロセスがコアから出たり入ったりし始めたならば，別のスケジューリングの機構を考え出さなければならない．

3.5 お わ り に

本章で述べたスケジューリング・アルゴリズムは，処理装置を待ち行列内のプロセスに効率よく割り当てる技法を広く紹介している．これらの規則は，また，オペレーティング・システムにおける他の資源割り当て問題に適用できる．多重レベル・スケジューリングの方法は，スケジューリング方式の実現が一定

の制限以下のオーバーヘッドだけで済むという例を与えている．これらの技法は知られているが，最適のスケジューリングはほとんどのシステムでは現状では実現不可能である．

どの方式が実現されようとも，その方式の有効性を確かめまた性能の障害となっている箇所を見つけ出す方法がなければならない．これはシステムの実現後，それを解析したり監視したりすることにより達成できるであろう．システムの性能評価の技法についての概括は，第9章で行なう．

演 習 問 題

3.1 先取りを使用する最高優先度先処理アルゴリズムを考えよう．そこでは，各プロセスの優先度は時間と共に変化するものとする．変化する比率は，プロセスがアイドルの時は定数 a，プロセスが実行中のときは定数 b であるとする．最高優先度のプロセスが2個以上あるときには，すべての最高優先度のプロセスは非常に小さい刻み幅を用いて，時分割形式に処理されるものとする．先取りは自由に行なえるものと仮定しよう．a および b をいろいろに変化させて，このアルゴリズムの動作を調べよ [Kleinlock, 1970].

3.2 ある特定のシステムにおけるプロセスのスケジューリング・アルゴリズムを記述せよ．記憶の割り当て方式がスケジューリング・アルゴリズムと何等かの点で関係しているか．

3.3 読者自身の身近にあるシステムにおいて，CPUが10%のアイドル時間を持つと，どの位の費用に相当するか．この費用と計算機設備の他の主な費用とを比較せよ．

3.4 先取りを許さない単一処理装置から成る機械があると考えよう．処理装置の時間への要求が事前にわかっているときに，平均の待ち時間を最小化するようなスケジューリング・アルゴリズムを考えよ．また，最大待ち時間を最小化するようなアルゴリズムを考えよ．先取りが自由に使えるときには，それらのアルゴリズムにどんな影響を与えるか．

3.5 多重プロセサ・システムにおいては，一つの処理装置に他のすべての処

理装置のスケジュールを任せるだろうか．あるいは，自分のジョブを別の処理装置に送り込んで，その処理装置が自分自身のスケジューリングを行なうようにするか．それぞれの方式の利点と欠点とを議論せよ．

3.6 理論を扱った多くの論文では，ジョブの到着間隔は指数分布に従うものと仮定している．読者のシステムにおいて，この仮定が正しいかどうかを確かめるための実験をせよ．

3.7 計算機システムに対する負荷は，一日中同じであるというわけではない．読者のシステムにおいて，一日の負荷の変動を求める実験をし，その結果をグラフに描け．負荷のピークをなだらかにするのに役立つと思われるような方法について概略を述べよ．

3.8 スケジューリングに対する解析的モデルでは，ほとんどの場合，多くの仮定をしている．たとえば，Sevcik [1971] のスケジューリングのモデルは，次のような仮定をしている：

1. 単一の処理装置上で，スケジュールを行なう．
2. スケジューリングの目的は，得られる処理を最大にする．
3. 各要求につけられた損失関数は，その完了の遅れによって生ずる利益の低下を反映している．
4. 各要求は要求クラスをつけられているが，各クラスに対しては処理時間の分布が与えられている．
5. 新しい要求が到着してくるときのプロセスが，各要求のクラスには知られている．
6. 未だ完了していない要求に対する先取りは，ある決められた大きさの処理装置の時間を必要とする．

実用的見地からみて，各々の仮定を批判し，また，正当性を議論せよ．これらの仮定に基づいて得られた解析的な結果を適用する前に，どんな種類の実験が必要であろうか．

3.9 システムの現在の状態に基づいて，負荷とジョブ・ミックスとを制御す

るようなスケジューリング・アルゴリズムを設計せよ．その動的な性質をシミュレーションによって調べよ．そのアルゴリズムは提供されるジョブによって生ずる環境の変化よりも速く，システムの不釣合いを修正することができると考えるか．その答の正当性を示せ．

3.10 腕木を用いて実現した相互排除を持つプロセスの集合を考えよう．危険部分を実行中のプロセスPが処理装置の制御権を放棄したときには，Pが再び制御権を得る迄は他の全てのプロセスは停止してしまう．危険部分の内部でプロセスが停止すること避けることができるようなスケジュールの方法について，概略を述べよ．同じ問題をメッセージの転送やモニタなど，他の相互通信機構を用いて考えよ．

第 4 章

記 憶 の 管 理

4.1 記憶管理の関数

記憶の管理は，3つの関数として考えることができよう［Denning ほか，1971］（図 4-1 参照）：

1. **名前付け関数**（naming function）（$\mathfrak{n}$）
   ユーザの定義した名前を，それが参照する情報の一意的な識別名へ写像する．
2. **記憶関数**（memory function）（$\mathfrak{m}$）
   一意的な識別名（すなわち，プログラムにおける番地）をそれがロードされる実際の記憶の位置（すなわち，記憶番地）へ写像する［Denning，1970］．
3. **内容関数**（contents function）（$\mathfrak{c}$）
   各記憶番地を，それが持っている値（value）へ写像する．

これらの3つの関数は，時間とともに変動する．すなわち，ジョブがシステムによって処理されている間に，それは変化するかもしれない．たとえば，$\mathfrak{n}$ は，一般にジョブがその参照しているシステムのモジュールやファイルに“結合（link）”されるまでは，決らない．$\mathfrak{m}$関数はジョブがロードされるときには，決定できる．しかし，ある種の割り当て方針の下では，ジョブがコアにいる間

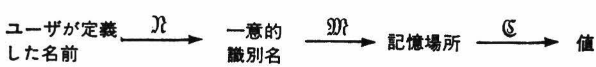


図 4-1 記憶の写像（$\mathfrak{n}$，名前付け関数；$\mathfrak{m}$，記憶関数；$\mathfrak{c}$，内容関数）

に$\mathfrak{m}$は変化する．あきらかに，関数$\mathfrak{c}$の結果は“格納”命令が実行されるごとに変化する．

4.1.1 束　縛

これらの写像の一つを固定化する操作を**束縛**（binding）という．写像fが束縛される時期を**fの束縛時期**（binding time of f）という．束縛はジョブのコーディングを行なうときに起るかもしれないし，また，ジョブがシステム内にいるとき，たとえばコンパイル時やロード時に起きるかもしれない．時には，写像の束縛が非常に頻繁に行なわれるので，それはもはや束縛されていないと考えた方が，ずっと自然であるようなこともある．内容関数$\mathfrak{c}$はそのようなたぐいの写像である．$\mathfrak{c}$は，完全にユーザによって指定されるのであるから，システムの観点から見れば特に興味のあるものではない．しかし，$\mathfrak{n}$と$\mathfrak{m}$とはシステムの制御の下にあるので，その束縛時期をどのように選ぶのかということは，システムにとっては重要な問題である．たとえば，記憶関数$\mathfrak{m}$を考えてみよう．$\mathfrak{m}$の束縛時期が遅ければ，記憶の使用効率を最大にするために，ジョブをコア内で動かすことができるので，柔軟性を持つためには望ましい．残念ながら，ある種の機械の構造においては，それを実現するために多くの費用を必要とする．実現は容易ではあるが，$\mathfrak{m}$の束縛時期を早くすることは，記憶の使用の釣り合いをとる能力に厳しい制限をつけてしまう．同じような考え方は，名前付け関数の場合にも適用できる．以後の議論では，読者は各々の実現方式において，柔軟性と費用との兼ね合いを常に考えているべきである．

4.1.2 名前付け

与えられた名前は，それが使われている環境によって，異った対象物を示すかもしれない．名前をそれが指し示す対象物と対応させることは，ソフトウェア・システムにおいてはよくあらわれることである．たとえば，**ブロック構造**（block-structure）を持ったプログラミング言語（たとえば，ALGOL）では，与えられた名前はそれが使われているブロックによっていくつかの異った変数を参照することができる．オペレーティング・システムにおいては，一般に関心のある環境はユーザ・プログラムの環境である．いくつかのユーザが，異っ

たファイルや手続きに対して，同じ名前を使用しても構わない．各ユーザの環境において，それぞれの名前に対して正しい対象物を対応させることは，オペレーティング・システムの仕事である．

一般に，異ったプログラムが同一の対象物を異った名前で参照できることが望ましい．これはファイルの場合に，しばしば助けになる．ファイルでは，異ったユーザが異った名前でシステム・ファイルを参照してもよい．一つの与えられた名前が，いろいろの場所で単一の対象物を参照するために用いられたときは，その名前があらわれたら，その対象物に結びつけることが重要である．前者の問題，すなわち，一つのファイルに対し複数個の名前を用いることに関しては，第6章で議論する．後者の問題，すなわち，名前をそれが参照しようとしている対象物と束縛することは次の節の主題とする．

4.1.3　一般的な結合の問題

いくつかの手続きとデータ構造とから成り立っているプログラムを考えよう．手続きやデータ構造のうち，いくつかのものはユーザが定義したもの（たとえば，FORTRAN や PL/I のプログラム）であり，他のものはシステムが定義したもの（FORTRAN のコンパイラや，ユーティリティ・ルーチン）である．各々は別々にコンパイル（あるいはアセンブル）したものであり，**オブジェクト・モジュール**（object module）というが，0番地からの相対的な番地がついている．一つのモジュールから他のモジュールの中の手続きを呼び出したり，データを使用したりすることができるように，モジュール同志で協力しあう．このような呼び出しや使用は**外部参照**（external reference）という．たとえば，一つのモジュールにある PROC 1 という手続きは，他のモジュールにある PROC 2 という手続きを呼び出そうとしてもよい．両モジュールは別々にコンパイルされるので，PROC 1 は PROC 2 の番地を“知る”ことはできない．PROC 1 は PROC 2 についてはただ名前を知っているに過ぎない．したがって，全体のプログラムがコア内にロードされる前に，この参照が**決定される**（resolved）必要がある．すなわち，PROC 1 はそのモジュール内で PROC 2 の番地を見つけ出さなければならない．参照の決定をするこのような過程を**結合すること**（linking）というが，それは次の3つのうちのどれかの方法で行なうこ

とができる：

1. プログラマが，コンパイル中に結合する手続きの集合を提供する．
2. 事前にコンパイルしたルーチンを，コアにロードする前かまたはその最中に結合する．
3. 結合は実行時迄遅らせ，その時にはシステムにより自動的に行なう．

大抵の高級言語では，方法1を実現して，ユーザが単一のプログラムの中でお互に呼び出すことのできるような手続きを定義できるようにする．したがって，手続き（PROC 1 や PROC 2）はコンパイラによって結合する．コンパイラによって生成するオブジェクト・モジュールは，コアに直接移すかまたは他のシステム・ルーチンに最初に結合する．モジュールを最初にシステム・ルーチンに結合しなければならない時は，結合方式は方法1と方法2との混合と考えることができる．

方法2においては，結合はコンパイル時からロード時の間に異ったステップとして行なう．結合とロードとは異ったステップとして実行できるけれども，一つのステップとしての**リンキング・ローダ**（linking loader）が，よく使われるようになってきた．外部参照を決定するほかに，リンキング・ローダは0から始まる相対番地をすべて修正して，そのモジュールがロードされる番地に合わせる．この操作は**再配置**（relocation）という．

実行時に結合する方法3は，プログラムを構成しているいくつかの手続きの名前が実行前にわからないときに役に立つ．このような状況は，データが実行する計算のタイプに影響を与える場合に生じてくる．必要になるすべてのルーチンを結合する代りに，それらが呼び出されたときにだけ結合する．この方法は第4．6節に示す割り当て方式と密接な関係にあるので，その記述はこの章の終りに行なうことにする．

4.2 結合の方法

結合方法の複雑さは，ほとんど，ハードウェアによって決まってしまう．この節では，単純なハードウェアの構造を仮定する．オブジェクト・モジュールの集団は0番地から相対的に番地付けした単一の**結合モジュール**(linked mod-

ule）になるように結合する．結合モジュールはそれをロードする位置とは無関係に番地付けするので，そのモジュールは**再配置可能**（relocatable）であるという．モジュールを主記憶にロードする時は，その最初の語の絶対番地が**再配置レジスタ**（relocation register）［または**ベース・レジスタ**（base register）］に格納される．こうすると，すべての番地の参照は，その参照の（相対）番地と再配置レジスタの内容とを加えることにより記憶内の語の絶対番地を得て決める．後の節では，他の番地付け機構と関連付けて，結合を議論する．

4.2.1 リンキング・ローダ

リンキング・ローダはオブジェクト・モジュールを結合して番地の再配置を行ない，そのプログラムを主記憶に物理的にロードすることを**1パス**（one-pass）で行なうルーチンである．実際のプログラムのテキストとその長さのほかに，各オブジェクト・モジュールはローダに対し次のことを示さなければならない：

1. モジュールで定義し，他のモジュールから参照する可能性のある名前．
2. モジュール内で参照しているが，そのモジュール内では定義していない名前（すなわち，外部参照名）．
3. そのモジュールが結合されたモジュールのどこに位置するのかに基づいて，再計算しなければならないような番地（たとえば，オブジェクト・モジュールの先頭から相対的にきめられる番地）．

ローダは，モジュールの外部を参照するような名前をすべて含んだような名前表を作ることから始める．表内の各名前には，2つの可能な解釈のうちどちらかを示すエントリが与えられる．その名前がすでにロードしたモジュールの中で定義されている場合は，表のエントリには結合したモジュールの先頭からの相対番地が入っている．もしも，その名前を定義しているオブジェクト・モジュールが，まだロードされていない場合は，表のエントリには最終的には名前に対する番地を持つと思われる番地フィールドの結合リストへのポインタが入っている．その名前が最終的に見つかったとき，結合リストを先頭から走査し，リストの各要素に番地を書き込む．各モジュールは，主記憶の連続領域に次々とロードする．番地の参照は，もはや，0番地からの相対的なものではな

く，前の方にロードしているモジュールの長さの和だけ下駄をはいていなければならない．次に名前表を更新し，モジュール内の各外部参照は決定するかまたはその名前に付随しているリストに結合する．全体のプログラムが一担ロードされると，名前表は壊すかまたはデバッギングのために退避する．

リンキング・ローダに関してもっと興味のある読者は，Gear［1969］や Donovan［1972］の本を参照すればよい．

4.2.2 リンケージ・エディタ

モジュール間の外部参照の解決や絶対番地の再配置は，ロードすることとは別の機能として分離することができる．前者の機能は**リンケージ・エディタ**（linkage editor）と呼ばれるが，オブジェクト・モジュールを混ぜ合わせて**ロード・モジュール**（load module）という単一のモジュールにする．再配置およびロードの処理は**ローダ**（loader）と呼ぶ別のルーチンによって行なう．

4.2.3 結合の方法の比較

2ステップの処理に対しては，いくつかの利点がある．第一に，リンケージ・エディティングはプログラムを実行する度毎に行なう必要がなく，実行のためのオーバーヘッドを軽くすることができる．第二に，コアの必要度に関する利点である．ローダはリンキング・ローダに比べてずっと単純なものであるから，ずっと小さいものになるであろう．したがって，実行ステップを開始するのにローダがコアに入っている必要があり，2ステップの処理の場合の方が少ないコアで済む．第三の利点は，リンケージ・エディタは4.4節に述べる“オーバーレイ“(overlay)”を実現できることである．

結合することとロードすることという2ステップの処理を用いるときの問題点は，入出力である．リンケージ・エディタはロード・モジュールを出力しなければならないが，それは次にはローダのために読み込まれる．ロードをする前には，大部分のユーザのプログラム・コードは，ほとんど修正する部分がない．したがって，しばしば最も高価な操作であるような読み込みや書き出しを用いることは，非常に無駄である．2ステップの処理の場合，最も節約になるのはプログラムを変更しない限り，結合はただ一回でよいという点である．し

かし，非常に少ない回数しか実行しないとか，非常に頻繁に変更を受けるようなユーザのジョブに対しては，のちほど実行するときに結合しないで済むという利点は，デバッギングの際に2ステップの処理に対して必要となる費用のために減殺されてしまう．したがって，ただ一回しか実行しないとか，しばしば変更するようなジョブにとってはリンキング・ローダの方が，一般に費用のかからないものとなろう．

以後，本章ではジョブは結合されて，0番地から始まる再配置可能なロード・モジュールとして存在するものと仮定する．再配置の方法は，使用する割り当て方式の形式によって異なる．この節で議論したような大抵の割り当て方式では，再配置はきわめて単純に行なわれる．したがって，結合したモジュールを，主記憶の中にどのように格納するかという点に，注意を集中することができる．

4.3　記憶の割り当て

記憶をジョブに割り当てる技法は**記憶割り当て方式**（storage allocation strategy）という．プログラムはその命令とデータとが主記憶の中にあるとき，はじめて実行できる．最も簡単な割り当て方式は，ジョブをそっくりそのまま連続したコアの領域にロードする方式である．しかしながら，ある時刻にはジョブのほんの一部分しか実行していないのであるから，現在アクティブになっているジョブの一部分だけをコアに格納することにより，記憶を節約することができる．ジョブの残りの部分は，それが必要になるまでドラムの様な周辺装置に置いておく．後者の方式では，記憶はずっと効率良く使用されるが，必要なときにジョブの一部分をコアに入れたり出したりするために，オーバーヘッドが生じてくる．以後の節では，ある条件の下では大きなオーバーヘッドを伴わないで，記憶の平均使用率が改善されるようないくつかの割り当て方式を紹介する．

4.3.1　記憶の写像の束縛

記憶関数$\mathfrak{m}$は，実際の記憶の位置を再配置可能なロード・モジュールに割り当てる．**静的割り当て**（static allocation）方式では，プログラムやコンパイラによって，ローダによってロードされる前かまたはロードされるときに，記憶

の位置を束縛する．第三の方式は，mを全然束縛せず，実行中にジョブが主記憶の内部を動きまわる方式である．この方式は**動的記憶割り当て**（dynamic storage allocation）と呼ぶが，システムは効率よく記憶を詰めることが可能となる．コアからは情報のブロックが出たり入ったりするので，ジョブの間には大抵使いものにならないような小さな空間ができてしまう．そのような隙間が多数あるときは，記憶は**断片化された**（fragmented）という．ロードされた後でもmが再び束縛できるならば，コアの中でジョブを動かして，これらの空間をまとめてジョブを格納できる位大きな領域にすることができる．したがって，動的記憶割り当てを用いれば，断片を少なくすることが柔軟に行なえる．

静的および動的割り当て方式に対する記憶の写像の束縛時期は，次の通りである［Denning ほか，1971］：

1. **絶対番地によるプログラミング**（absolute address programming）
   プログラマは，プログラムを記述するときに，機械の番地を指定する．
2. **記号によるプログラミング**（symbolic programming）
   コンパイラやアセンブラはその翻訳過程において，記号名から記憶番地を生成する．これらの記憶番地は，次のように割り当てる：
   (a) 生成した記憶番地は，固定した絶対番地である．
   (b) 記憶番地は再配置可能であり，そのモジュールを実行するときに，ローダによって絶対番地を割り当てる．
   (c) 記憶番地は再配置可能であり，参照するごとに絶対番地を割り当てる．

方法1，2a，2bは，ジョブをコアにロードする前またはその間に記憶の写像を束縛するという意味で，静的なものである．方法1は，写像mとnとをただ一回だけ結合する場合の例である．この方式は，一時に，一つのジョブだけしか実行しないような小さな機械の場合には，非常によく使う．方法2は，mとnとの束縛を分離して行なう．方法2aでは，mはnのすぐ後で束縛する．高速のコンパイラはよくこの方法を使う．すなわち，実行の能率を犠牲にして，コンパイルとロードの速度を上げている．生成した命令を実行可能な位置に格納するので，結合したりロードするためのオーバーヘッドは，ほとんどなくなる．

方法 2b は，IBM/360 シリーズの大部分のモデルで用いられているが，𝔪はロード時に束縛する．この方式では，ロード・モジュールは再配置可能となっている．すなわち，それはコアのどこにおいても実行できる．しかし，一般には最初にロードした後では移動しない．以上の３つの方法は，簡単に実現できる．

方法2cは，𝔪を束縛することがないので，まさしく動的なものである．この方法を用いる技法については，本章の後半で述べる．

情報の再配置が行なわれる場合，回転する装置を用いるときのオーバーヘッドを最小にするためには，情報をブロックにして移動するのが最適である．第1．1節に述べたように，ディスクやドラムからレコードを捜し出してくるとき，**シーク時間**（seek time）が重要な鍵となる．すなわち，レコードを捜す時間は，一般に，それをコアに転送する時間に比べてずっと大きく，ブロックの長さは相対的に重要ではなくなってくる．したがって，情報を大きなブロックで転送する方がずっと効率的である．シーク操作の回数を減らすことにより，長いブロックを転送する時間を充分補うことができる．

実際には，ジョブをどのようにブロックに分けるかということは，割り当て方式によって左左される．ブロックは論理的な集団すなわち手続きなどに対応してもよいし，また物理的な装置の特性，すなわちドラムのトラックの大きさに対応していてもよい．以後，本章では，明示していない限りはブロックは任意の大きさでありまたジョブは再配置可能の形をしていると仮定する．

4.4 オーバーレイ化

２つのプログラム・モジュールが同時にはコアに存在する必要がなく，また値を退避することを注意深く行なっていれば，両方を同じ記憶場所に格納することができる [Pankhurst, 1968]．すなわち，記憶の写像𝔪はロード時に束縛するが，複数個のプログラム番地が同一の記憶番地に写像することを許している．たとえば，Ｃを主手続きとし，それは最初手続きＡを呼び出し，次にしばらくしてから，手続きＢを呼び出すものとする．**図4-2** に示すように，ＡとＢとは同一の記憶空間を占有することが許される．

結合したモジュールに対して単一の再配置レジスタを使うようなシステムにおいては，手続きＣが CALL を実行したとき，正しい手続きが40Ｋ－70Ｋ の

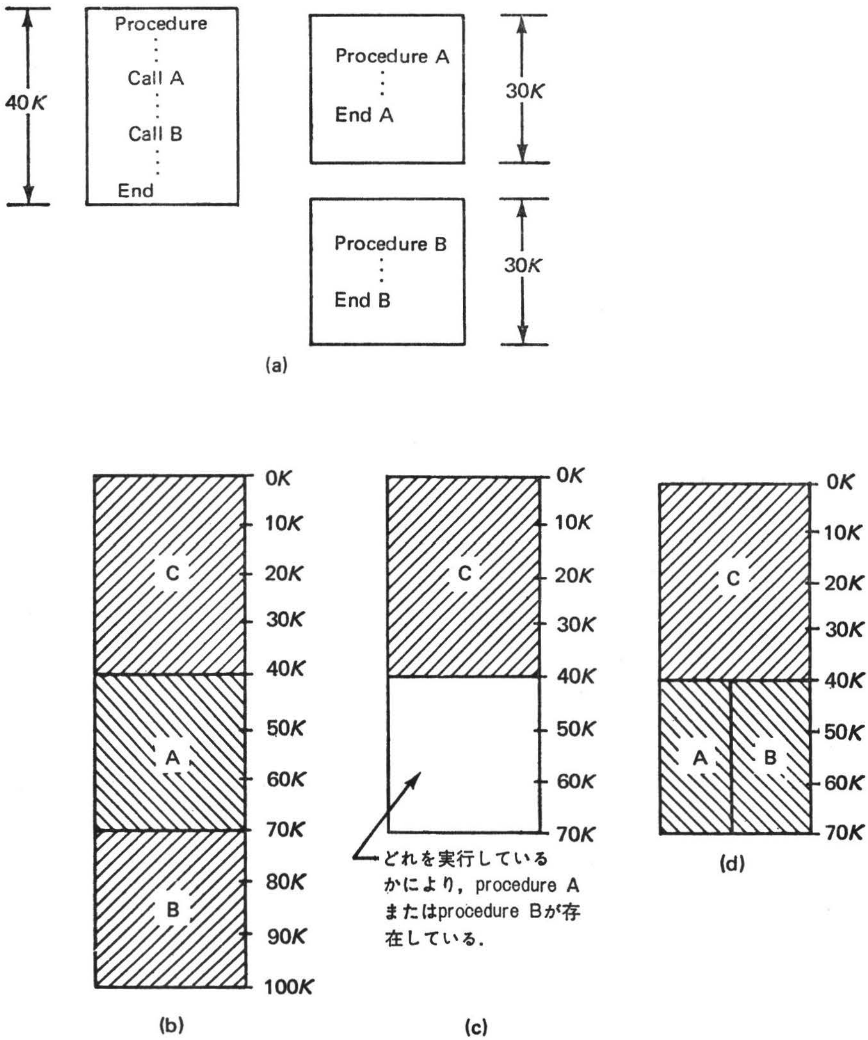


図 4-2 記憶のオーバーレイの例. (a) お互に呼び合う手続きの集合； (b) 連続的に格納された手続き；(c) 40K—70K にオーバーレイされる手続きAとBとを持った手続き； (d) 同一の記憶空間40K—70Kを占有する手続きAとBとを示す“オーバーレーイ・マップ”.

コア位置に存在していることを保証するのは，一般にはリンケージ・エディタおよびローダの責任である．リンケージ・エディタに対しては，プログラム・モジュールと，どの手続きが主記憶空間を共有するかを示す**オーバーレイ・マップ**（overlay map）（**図 4-2** d）とを入力する．そこで，リンケージ・エディタはオーバーレイされているどれかのルーチンが呼び出されたとき，ローダを呼び起すような特別のコマンドを作り出す．ローダは呼ばれたモジュールがまだロードされていないときは，それを主記憶に持ってくる．したがって，プログラム・モジュールは，実行中必要なときに，動的に主記憶に入ったりまたそこから出たりする．

モジュールをオーバーレイすることにより，プログラムによって必要となる主記憶の量を，かなり多く減らすことができる．オーバーレイを許しているシステムにおいては，全体の大きさが主記憶の大きさを越えてしまうようなプログラムでも実行をすることができる．たとえば，80Kのコアを持ちローダを実行するために10K必要となるようなシステムを考えてみよう．このシステムにおいては，ジョブの全長はコアよりも大きいけれども，**図 4-2** に示したようなジョブを実行することができる．したがって，オーバーレイを許すような複雑なリンケージ・エディタやローダを作ることにより，記憶の割り当てを柔軟に行なうことができる．

しかしながら，オーバーレイによっても解決できないような記憶管理の問題が，まだ2つある．最初に与えた仮定，すなわち，オーバーレイされているモジュールは同一時間にコアに存在する必要がないという条件を，もう一度考えてみよう．データに依存する手続きの呼び出しとか，再帰的手続きのようなある種の動的なプログラムの動作に対しては，2つの手続きがオーバーレイできるかどうかを事前に予測するのは不可能である．すなわち，プログラマは2つの手続きがお互に呼び合うかどうかを知ることができず，したがってオーバーレイ・マップを作ることができない．リンケージ・エディタに対しオーバーレイ構造を示すのは，プログラマの責任であるので，プログラムの動作を予測できなければオーバーレイは実現不可能となる．

これまでの議論では，ジョブが使用できる主記憶空間の量は事前にわかっているという仮定をしてきた．実行中に使用可能な記憶の量は，動的に変化して

もよいが，これはオーバーレイにおける第2の重要な問題を提起する．システムの負荷がふえると，記憶はどんどん混んでくる．空間の割り当てが少ないと，ジョブのオーバーレイ構造はその記憶の割り当て部分にはもはや適合しないかもしれない．しかし，他のジョブに属している手続きで，最初のユーザによってオーバーレイできるようなものがあるかもしれない．プログラマは，事前にこれを知ることはできない．逆に機械への負荷が軽いときは，動的なオーバーレイが引き起す余分なオーバーヘッドは必要ないかもしれない．使用できる記憶は，充分あるからである．重要なことは，プログラムを実行するときのシステムの状態に応じて，異った方式で記憶の管理をしなければならないことである．さらに，システムはジョブ間において，大域的に記憶の割り当てを最適化することができなければならない．単一のユーザはシステムの負荷や他のプログラムの動作などを予測できないので，動的なオーバーレイ方式も全体の記憶の使用を改善するためには，あまり役には立たない．

4.5 ジョブのスワッピング

オーバーレイ方式においては，ジョブが，一旦記憶領域を割り当てられロードされると，それが完了するまでその領域内で実行をする．すなわち，記憶の写像$\mathfrak{m}$はロード時に束縛する．ジョブの記憶の最大要求量と使用可能な主記憶の量を予測するという問題のほかに，この仮定は2つの好ましくない効果をもたらす．第一のものは，記憶の使用効率に関係したものである．**図 4-3a** に示したように，パックされた記憶を持つ多重プログラミングの環境を考えてみよう．JOB3が最初に完了すると，記憶マップは**図 4-3b** のようになる．150Kの記憶は未使用の状態ではあるけれども，記憶が断片化しているので，150Kの連続領域に対する新しい要求をシステムは満足させることができない．JOB2を50K下へ移動して，JOB1と連続になるようにすれば，あきらかに一つの解になる．しかしながら，こうするためには，ロードした後でも記憶の写像を再び束縛する必要がでてくる．

第二の好ましくない効果は，記憶以外の資源の使用効率に関してである．再び，**図 4-3a** に示したように，3つのアクティブなジョブのあることを仮定してみよう．JOB2だけが，ディスク駆動装置を現在使用しているジョブとしよ

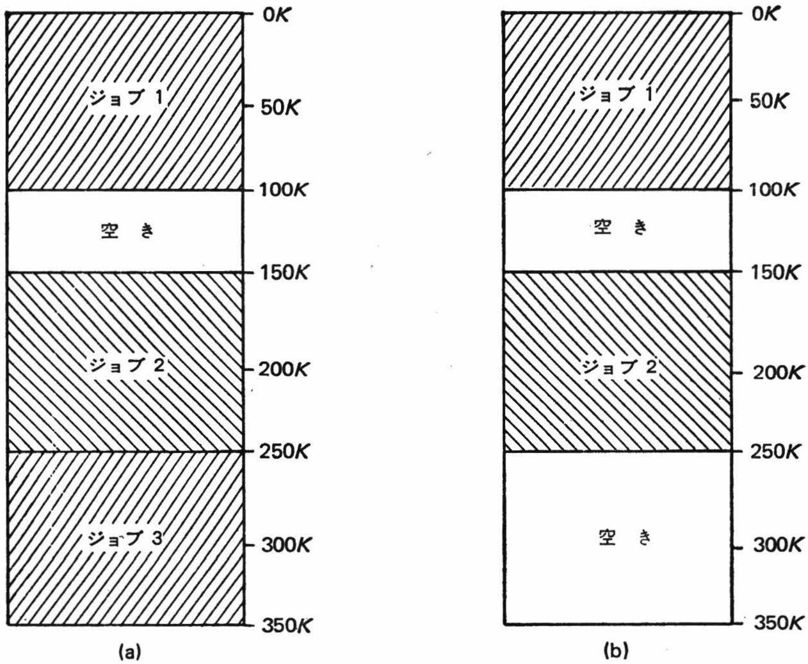


図 4-3 記憶の断片化の例. (a) 記憶の最初の状態；(b) JOB 3が終った後の記憶の状態

う. JOB 2 がその実行を終了したときは，これらのディスク駆動装置の支配を放棄し，コアから去ってゆく. あきらかに，ディスク駆動装置はディスクを使用する新しいジョブがコアに入って来るまでは，アイドル状態になっている. 実行を待っていてディスクを必要とするジョブ（JOB 4 とする）は，250Kの大きさであると仮定しよう. 残念ながら，150Kしか空いていないので，JOB 4 に対しては足りない. 充分大きな空間が空くまでディスクをアイドルにしておくのを避けるためには，まだアクティブであるようなジョブ，すなわち JOB 1 や JOB 3 を主記憶から動かして，JOB 4 に対する空間を作った方がよいだろう. JOB 4 をすみやかにコアに持ってくれば，ディスクの使用効率が上ることになる（すなわち，アイドル時間の総和を減らす). しかしながら，記憶の分布が都

合の悪い場合と同様に，この方式はロードした後でもジョブを動かし記憶の写像を再束縛する必要がでてくる．

前述した困難な問題を解決する一つの方法は，最初にロードした後，コアから（またはコアの中へ）全体のジョブを動かすことである．この方式は**スワッピング**（swapping）または**ロール・イン・ロール・アウト**（roll-in-roll-out）というが，これは再配置レジスタを用いることによって実現できる．前にも注意したように，全てのジョブは0番地を基準にして，番地付けしている．番地の計算において，ジョブの参照する各々の番地と再配置レジスタの内容とを加えることにより，ジョブの番地付け方式を破壊することなく，コアの新しい位置に動かすことができる．すなわち，ジョブがA番地から始まる記憶内に存在しているときは，値Aを再配置レジスタに格納する．

ロードした後でもプログラムを再配置できるならば，記憶の管理や資源の使用効率の釣合いをとるときに大きな助けとなる．第一に，**図4-3**に示すように，断片化によって連続的な記憶領域の大きさが小さくなったときに，記憶を再パックすることができる．第二に，システムは，現在どのジョブをコア内に入れるかということを支配することができるので，システムのほかの部分をもっと有効に使うことができる．たとえば，コアに存在しているジョブがすべて**入出力限界**（I/O bound）になっているとき，負荷の釣合いを取るために，その中の一つをスワップ・アウトして，**計算限界**（compute bound）のジョブを優遇してもよい．ただし，スワッピングによって得た節約が，ジョブを動かすために生ずるオーバーヘッドによって相殺することがないように気をつけなければならない．

記憶からプログラムをスワップ・インしたりスワップ・アウトすることは，あきらかに時間のかかる操作である．スワッピングのオーバーヘッドを減らす一つの方法は，リエントラントのコードを用いることである．いくつかのジョブが同時に使いたがるような大きなプログラム，たとえば，コンパイラなどが完全に分離したデータ・セグメントを持った**純手続き**（pure procedure）として書かれているときは，一時には，そのコピーをただひとつコアの中に入れておくだけでよい．複数個のジョブは自分自身のデータ領域を持って，その一つのデータを使うことができる．この手続きを使用するジョブに対しては，それ

ぞれのデータ領域だけをスワップの対象とすればよい．リエントラント・コードは，他の記憶方式に対しても有効に働く重要な概念である．しかしながら，多数のジョブ間である決められたブロックが頻繁に共有される場合だけ，スワッピングのオーバーヘッドを大きく減らすことができる．

スワッピングにおいて基本的に困難な点は，ジョブ全体をコアから追い出さなければいけないということである．これは，一般に，実際に必要なものよりもずっと大きく見積ってある．ジョブがその手続きのうちの一つを実行しているときは，その他の手続きのうちの大部分のものは，少なくとも一時的にはコアの中に存在している必要はない．したがって，ジョブ全体を消さず，その実行も邪魔をしないで，余分の隙間を作ることができる．オーバーレイの背後にある考え方は，ジョブの一部分をロール・アウトするということである．オーバーレイにおける問題点は，プログラマはどのコードを安全に掃き出してよいかということを予測しなければならないということである．もっとよい解決法は，スワッピングの動的な性質と，一時には，ジョブの一部分をスワッピングするというオプションとを結びつけることである．以後のこの章では，この目的を達成する方式を主題とする．

4.6 セグメント化

大抵のプログラムは多数の論理的な単位，たとえば手続き，データ領域，プログラム・ブロックなどから構成されている．**セグメント化**（segmentation）はプログラムの番地付けの構造が，プログラムの論理的な分割を反映するような方式である．セグメント化においては，各プログラムの番地空間を**セグメント**（segment）と呼ぶ多数の可変長のブロックに分割する．それは，一般に，プログラムの論理的な分割に対応している．たとえば，セグメントは手続きやデータ領域であってよい．プログラムの中の語は，セグメントの名前とそのセグメント内の語の位置，すなわち**偏差**（displacement）との2つの部分から成る番地によって識別する．プログラムのセグメントは名前によって参照するから，記憶割り当て機構では，主記憶の連続的でない位置にセグメントを格納することができる．実際，プログラムのすべてのセグメントが，一時に主記憶の中に存在する必要はない．すなわち，それらのうちのいくつかは周辺機器に存

在していることができる．プログラムが主記憶にないセグメントを参照しようとすると，システムが介在してそのセグメントを二次記憶から持ってくる．セグメント化されたプログラムの番地空間はセグメント化されていないものに比べて，ずっと柔軟性に富んではいるが，特別のハードウェアがないと２つの部分から成る番地の実現はきわめて費用がかかる．したがって，セグメント化を用いる大抵のシステムでは，番地の計算のためにハードウェアの助けを借りている．

4.6.1 セグメント化の実現

システム内の各番地の参照は対 [s, d] で行なうが，ここで s はセグメント名，d は偏差である．ジョブやプロセスには，そのジョブに属する各セグメントに対して一つづつのエントリを持った**セグメント・テーブル** (segment table) を，コア内に常駐の形で与えてある．この表はプログラムの番地を実際の記憶装置の位置に写像するためにシステムが使用するが，それは**セグメント・テーブル・レジスタ** (segment table register) と呼ばれるハードウェアのレジスタによって指されている．セグメント・テーブルの s 番目のエントリには，次のものが格納してある：

1. **フラグ** (flag) [または**存在ビット** (presence bit)]：s 番目のセグメントが，コア内に現在存在しているかどうかを示す．
2. s 番目のセグメントの**ベース・アドレス** (base address).
3. そのセグメントによって占められている語の数を示す**上限値** (limit).
4. アクセスのモードをチェックする保護ビット（オプション）.

語 [s, d] を使用するときには，セグメント・テーブル・レジスタを介してセグメント・テーブルに入ってくる(**図 4-4**)．テーブルの s 番目のエントリは，記憶内のセグメント s の位置を示している．このセグメント内の d 番目の位置が，望んでいる語 [s, d] である．セグメント・テーブルへの１アクセスとセグメント内の語への１アクセスと２つの記憶アクセスが語 [s,d] を得るためには必要となる．セグメントは大きさが変るので，d に対する固定した制限値はない．したがって，上限値の情報はセグメントの端を越えてジョブがアクセス

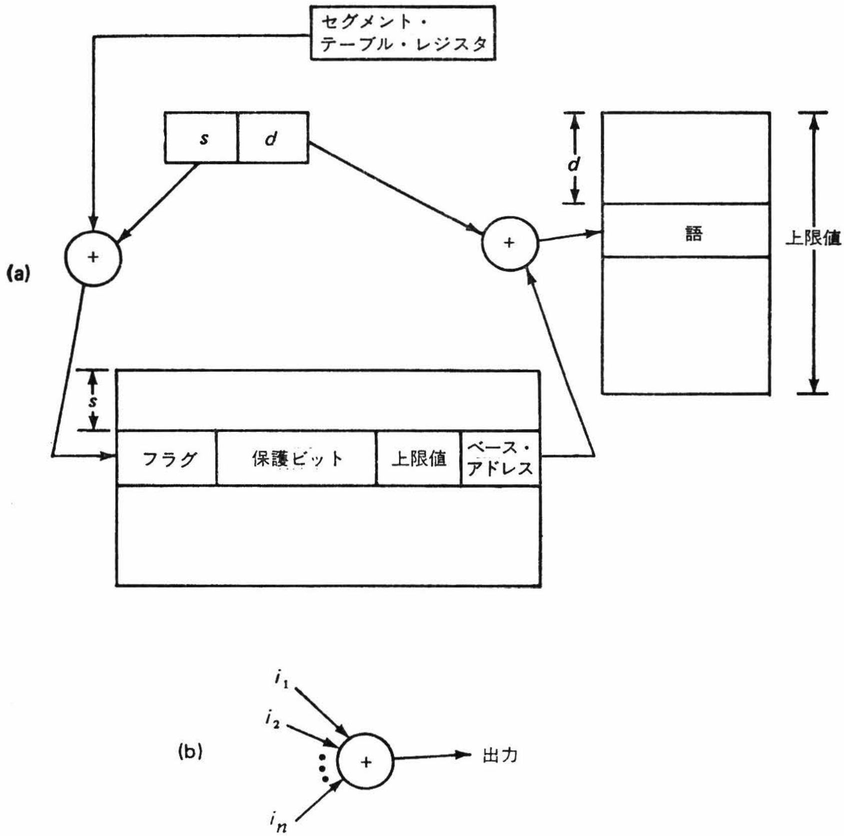


図 4-4 セグメント化に於ける番地の計算 [(b)における記号は，出力$=i_1+i_2+\cdots+i_n$となるような加算器を示す．c(X)を，語Xの内容とする．このとき，c(語)$=$c (c (c (セグメント・テーブル・レジスタ) $+s$)$+d$)]

することを防ぐために必要となる．多くのシステムたとえば Burroughs B 5700 では，上述した番地の計算と上限値のチェックはハードウェアの助けを借りて行なっている [Organick, 1973].

番地の計算を行なう前には，フラッグをチェックしてセグメントがコア内に存在するがどうか確かめる．もしそうであれば，記憶の番地は前のパラグラフで示した方法で自動的に計算される．もしも，セグメントがコア内に存在しな

ければ**セグメント・フォールト** (segment fault) が生ずることになり，ハードウェアの割り込みが起こり，**セグメント・フォールト・ハンドラ** (segment fault handler) と呼ぶスーパーバイザ・ルーチンを呼び出す．セグメント・フォールト・ハンドラは，二次記憶上に必要なセグメントを見つけ出し，それを主記憶に転送してくる．新しいセグメントに対して主記憶上に空きがなければ，システムはコアに存在しているセグメントを周辺装置に移して，空きを作り出せばよい．

任意の時点において，システムが他のプログラムに切り替えをしたいときには，セグメント・テーブル・レジスタの値を新しいセグメント・テーブルを指すように変更し，[s, d] による参照は新しいテーブルを用いて解釈する．

4.6.2 セグメント化の利点

セグメント化したプログラムの番地空間は，従来の絶対番地方式に比較して多くの利点を備えている．第一に，オーバレイはプログラムが事前に何も指定しなくても，システムが実行してくれる．プログラムのすべてのセグメントに対してコア内に充分な空きがなければ，一時的にいくつかのセグメントを二次記憶に掃き出す．オーバーレイとは異なり，セグメント化においてはプログラムはお互に相手の記憶領域のオーバーレイを行なうことができる．プログラムが新しいセグメントを主記憶に転送してくる必要が生じたときには，システムは主記憶内の任意のセグメントを二次記憶に掃き出すことができる．置き換えられるセグメントは，コアの中に新しく転送してきたセグメントと同一のプログラムに属している必要はない．セグメントを二次記憶に掃き出す際に，適当なセグメント・テーブル内のフラッグをオフにしている限りにおいては，そのセグメントがどのセグメント・テーブルに所属しているかということは，問題ではない．

セグメント化の第二の利点は，結合に関することである．セグメント化された番地空間においては，オブジェクト・モジュールはセグメントとしてプログラムすることができる．結合ルーチンは絶対番地にとらわれなくてもよいので，外部参照の解決はかなり簡単化される．実際，結合操作は外部参照に対応する [s, d] 番地を表内で見つけるという仕事に帰着される．ロードしてくる操作

は，いまや各セグメントを指すようにセグメント・テーブルを初期設定するということだけで済んでしまう．さらに，オーバーレイはシステムが行なってくれるので，オーバーレイのための特別なローダやリンケージ・エディタは，もはや必要がなくなる．

セグメント化はリエントラント・コードの共有を可能にする．独立したセグメントを持つリエントラント型手続きは，その場所をいくつかのセグメント・テーブルに登録することによって，共有することができる．そのセグメントをコアや二次記憶内の新しい場所に移したときは，すべてのテーブルの変更を特に注意深く行なわなければならない．セグメントの共有の例については，後節で述べる．

ハードウェアがセグメント化された番地空間を実現しているようならば，前述した利点の大部分は，ユーザの見地からすれば，“関係はない”．複雑な番地参照の機能は，システムが実施してくれる．すなわち，ユーザは自分のセグメントが連続していないとか，必ずコアに存在しなければならないなどということには無関心である．しかし，セグメント化によって，セグメントがコアと周辺装置の間を効率よく転送できるように記憶を管理するという問題が生じてくる．セグメントの管理の技法については，第5章で議論する．

4.7 ページ化

ページ化（paging）とは，記憶番地空間を固定長のブロックに分割するという記憶の管理の技法である．ページ化においては，コアの記憶空間を**ページ・フレーム**（page frame）と呼ぶ同一の大きさの物理的な部分に分割する．プログラムとデータは**ページ**（page）と呼ぶ論理的な単位に分割するが，それはコアのページ・フレームと同一の大きさである．したがって，1ページの情報は1ページ・フレームにロードすることができる．

4.7.1 ページ化の実現

ページ化した記憶空間における番地の参照は，セグメント化したプログラムの番地空間の場合に似ている．各参照は対［p, d］で与えるが，ここにpはページの名前，dはページp内の偏差である．

各ジョブに対しては，そのジョブを構成するようなページのリスト［**ページ・セット**（page set）と呼ぶ］を持った**ページ・テーブル**（page table）を与えている．ページ・テーブルの各エントリはそのジョブの一つのページに対応し，次のような情報を持っている：

1. そのページが，コア内に存在しているかどうかを示すフラッグ．
2. そのページの場所（コア内または二次記憶内）．
3. アクセスのモードをチェックするための保護ビット（オプション）．

さらに，**ページ・テーブル・レジスタ**（page table register）と呼ぶ特殊なレジスタがあり，それは現在実行中のジョブに与えられているページ・テーブルのコア内の場所を示している．

以後の番地計算においては，読者は**図 4-5** を参照されたい．実行中に番地［p, d］がでてくると，ページ・テーブル・レジスタを用いてページ・テーブルを参照する．ページ・テーブルの p 番目のエントリは，必要とするページへのポインタを含んでいる．そのページの先頭から d の大きさの位置のところに，要求した語が見つかる．したがって，一語を得るために記憶への 2 つのアクセスが必要となる．すなわち，ページ・テーブルへのアクセスおよびページ自身へのアクセスである．セグメント化の場合と同様にして，ページ・テーブルに対する特殊なハードウェアを設けて，この参照によって引き起される遅れを少なくすることが可能である（たとえば，MULTICS システム［Organick, 1972］）．

ページへのポインタに付随して，そのページがすでにコア内に存在しているかどうかを示すフラッグがある．番地計算を行なう前に，ハードウェアによってこのフラッグのチェックを行なう．もしも，これが *on* になっていれば，そのページはコア内に存在し，番地は自動的に計算される．もしも，そうでなければ，ハードウェアにより割り込みを引き起し，ページをコア内に持って来なければならないことをスーパーバイザに知らせる．これを**ページ・フォールト**(page fault)という．ポインタとフラッグのほかに，保護ビットもあるかもしれない．これらのビットは，要求されているアクセスのモード（たとえば，読みまたは書き）が許されているかどうかを決定するために用いる．番地の計算の場合と同様にして，保護ビットのチェックはハードウェアの助けを借りて行な

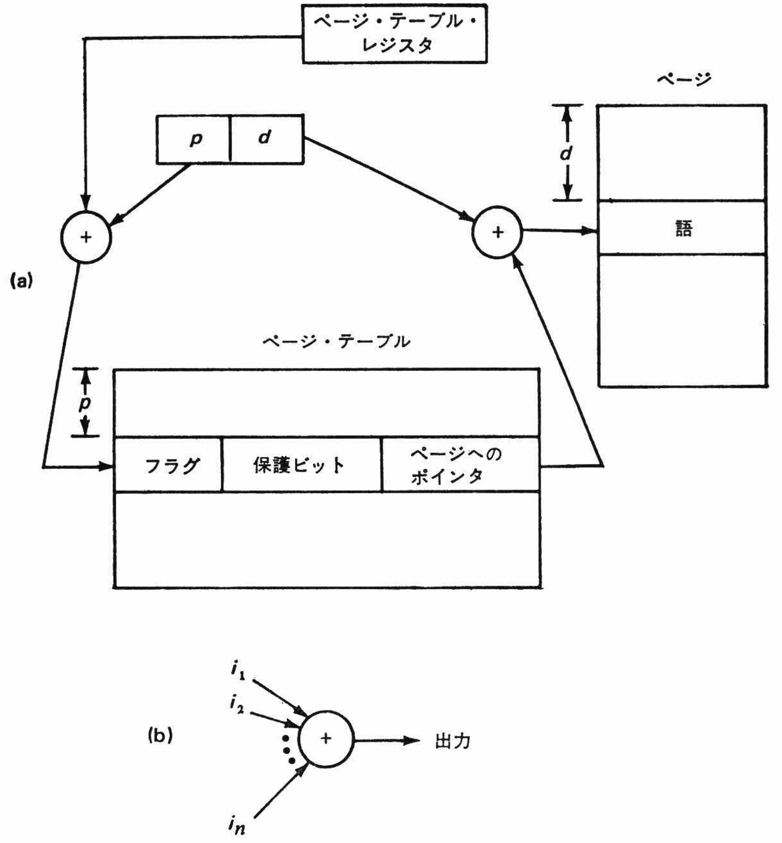


図 4-5　ページ化における番地の計算［(b)における記号は，出力$=i_1+i_2+\cdots+i_n$となる．加算器を示す c (語)＝c（c（c（ページ・テーブル・レジスタ）$+p$)$+d$)］

うことができる．

各々のページ化・システムでは，**フェッチ則**（fetch policy）と**ページ置き換え**（page replacement）の方式とを採用しなければならない．前者の規則とは，コア内に持ってくるために補助記憶からページを選択する方法を指す．後者は，あるページをコア内に持って来る準備ができたときに，折り重ねてしまうページの選択の方法を指す．**ページの移動量**（page traffic）は，ページがコ

ア内に入ったり出たりする動きの量であるが，それはフェッチ則と置き換えの方式の関数となる．フェッチと置き換えの規則は，セグメントを二次記憶から主記憶に持って来なければならないようなセグメント化された番地空間に対しても，同様に重要な問題である．ページやセグメントをフェッチしたり置き換えたりする方法は，第5章で解析し議論する．

4.7.2 ページ化の利点

セグメント化したプログラムの番地空間における多くの利点は，ページ化した記憶の番地空間においても存在する．セグメント化におけるオーバーレイについて前にのべた論点は，ページ化に対しても全く同様にあてはまる．セグメント化における場合と同じように，ハードウェアの助けを借りて番地の計算をすれば，ユーザに対して番地付けの複雑さをすっきりしたものにしてくれる．しかしながら，固定長のページを用いるということによって，2つの技法に重要な差異をもたらす．

ページの大きさは固定されているので，一ページの中にはプログラムの論理的な分割部分全体が入りきらないかもしれない．したがって，ページ化した番地空間に於ける結合は，セグメント化における場合ほど簡単なものではない．しかし，固定長のページを用いれば，記憶の割り当てが非常に簡単になる．すべてのページは同じ大きさなので，新しいページをコア内に持ってくることは，新しいセグメントを持ってくることよりも，ずっと容易に行なえる．そのページは空いたページ・フレームに置くか，または新しいページの空間を作るためにどれか一つのページを置き換える．どちらの場合も，新しいページに対して空きを作るために，コア内のページの再配置をする必要はない．

4.7.3 断 片 化

断片化という言葉は，記憶内の隙間を意味する．**外部断片化**（external fragmentation）とは，記憶のブロックの間に空白部分があらわれることを指すのに対し，**内部断片化**（internal fragmentation）とは，ブロック自身の内部に空白部分があらわれることを指す．固定長のブロック（すなわちページ）を用いた場合は，ブロックが追い出されるときは同一の大きさのものと置き換わるの

で，外部断片化は生じない．したがって，ブロック間には使用不可能な隙間は存在しない．しかしながら，内部断片化は生ずる．何故ならば，情報の論理的なグループ分けは，ページの大きさによってきっちりと割り切れないことがあるからである．たとえば，一つのグループは3ブロックと半分を占め，半ブロックは空にしておくようなことが生ずる．それゆえ，一つのページには一つ以上のジョブのデータや命令を含ませないという方針を用いるとすれば，各ジョブにおいて少なくとも一つのブロックは一部分しか使用していない．逆に，可変長のブロックを用いれば，情報の長さに合うようにブロックの長さをあつらえることができるので，内部断片化の問題は生じない．残念ながら，この方式を用いると一つのブロックを記憶から追い出したとき，同じ長さのもので置き換えることが出来ないかもしれないので，外部断片化を生じてしまう．したがって，ブロックがコアに入ってきたり出ていったりする間に，隙間は大きくなってゆく．断片化の様子を，**図4-6**および**図4-7**に示す．

オペレーティング・システムがページ化した環境で働く場合，ページの大きさを決めるときには，ページの**移動量**（traffic）と内部断片化とを考慮に入れ

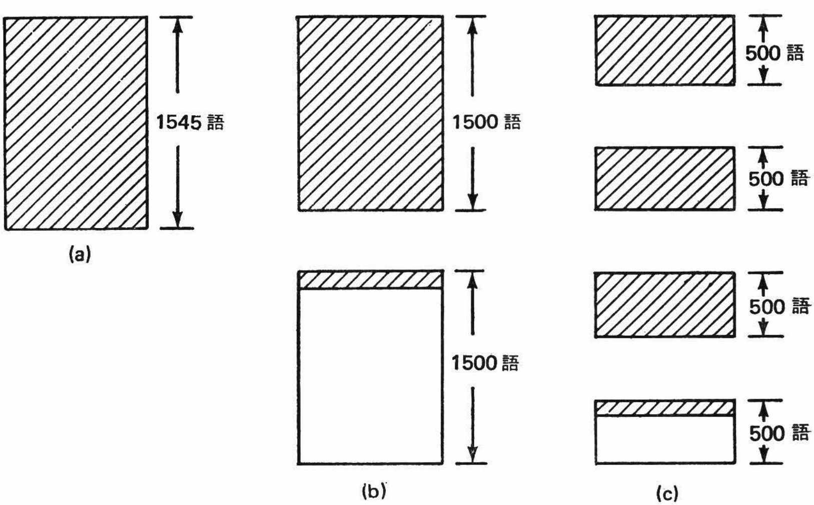


図 4-6　内部断片化．(a) ジョブを持ってくる；(b) ジョブが1500語のページに分割され，1455語が無駄になる；(c) 500語のページを用いた場合，455語が無駄になる．（ジョブをページに分割する場合，いくらかの空間が無駄になる．一般に，ページの大きさが小さければ，内部断片化は少なくなる．）

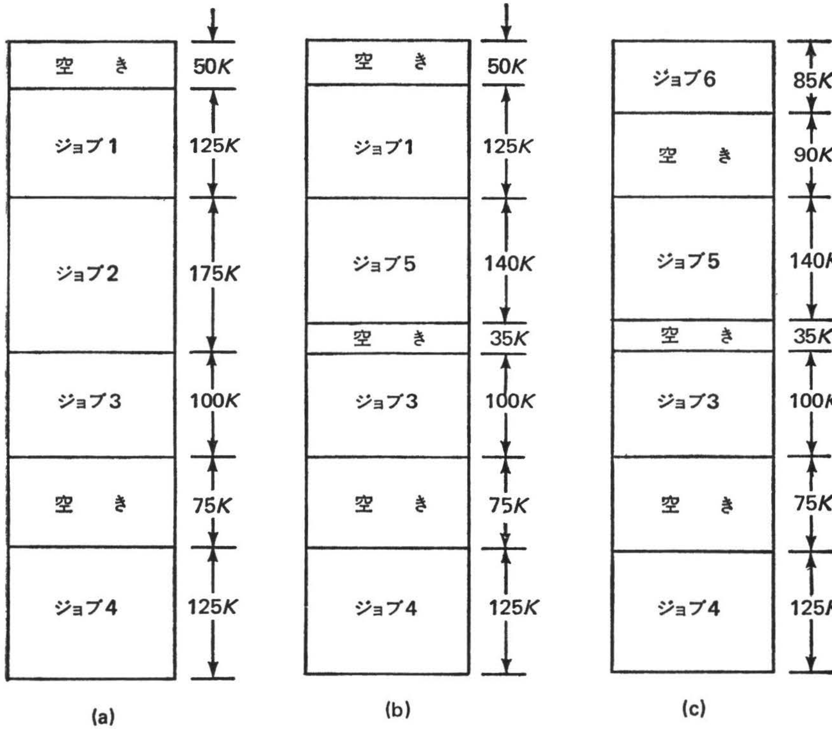


図 4-7 外部断片化. (a) 記憶の初期状態；(b)ジョブ2が去り，ジョブ5が到着する；(c) ジョブ1が去り，ジョブ6が到着する．（可変長のジョブが到着したり，去ったりするにつれて，記憶内に，隙間が生じたり，消滅したりする．）

なければならない．大きなブロックを一つ動かす方が小さなブロックを多数動かすよりもずっと能率的であるから，上限になるべく近づけて大きめのページを用いた方が望ましい．大きなページを用いれば，一つのジョブを格納するのにほんの少しのページを用いるだけでよいことになる．システムの中に少数のページしかなければ，コアから移動するページの数は少ないことになるので，ページの移動量は低いことになる．しかしながら，大きなページを用いると大量の内部断片化が生じることになる．たとえば，1545語の大きさのジョブを考えよう（**図 4-6** 参照)．もしも，ページが1500語の大きさであれば，2ページ必要になる．しかし，二番目のページでは，1455語が無駄になってしまう．ペー

ジが500語の大きさであれば，4ページ必要になる．こうすれば，ページの移動量の期待値は増加するが，455語しか無駄にはならない．言い換えれば，ページの移動を処理するためのオーバーヘッドと内部断片化との間に釣合いが必要ということである．両方の要因を，できるだけ小さくするような妥協点としてのページの大きさを見つけ出すことが目標となる．現在，一般的に用いられているページの大きさは，大体256～1024語である．

4.8 ページ化機構を持ったセグメント化

セグメント化とページ化は一緒に実現して，各セグメントは自分自身のページ・テーブルを用いてページ化するようにしてもよい．この方式では，一つの番地は3つの要素［s, p, d］から成り立つ．第一の要素は，セグメントの番号sを示す(**図4-8**参照)．セグメント・テーブルは，セグメント・テーブル・レジスタを介して参照する．セグメント・テーブルのs番目の場所には，そのセグメントに対するページ・テーブルのベース・アドレスと，その長さ（すなわち上限）とが書いてある．番地の第二の要素pは，セグメントsのp番目のページを示すページ・テーブルのエントリの場所を与える．ここで偏差dはページのベース・アドレスに加えられて，語を見つけ出す．したがって，番地［s, p, d］はセグメント・テーブル・レジスタの内容によって定義されるジョブのs番目のセグメントの，p番目のページの，d番目の語というように解釈する．

このような3要素への参照を行なうためには，3つの**記憶サイクル**（memory cycle）が必要になるが，これはハードウェアを採用して番地付けをするにしても，堪え難い遅れを生ずることになる．この問題は**連想レジスタ**（associative register）という高速の記憶セルの集団を用いて，sとpの組み合せをそれに対応するページの物理的な場所に写像することによって解決している（**図4-9**参照）．連想レジスタの主要な性質は，それらが，並列的に走査できるということである．† ［s, p, d］への記憶の参照が生じたとき，［s, p］が連想レジスタに

† **訳注**．与えられた内容を持つセルがあるかどうかを，ハードウエア的にパターン照合をしながら捜すことができる．その意味で，連想レジスタは**内容番地付け可能記憶**（content addressable memory）ともいう．

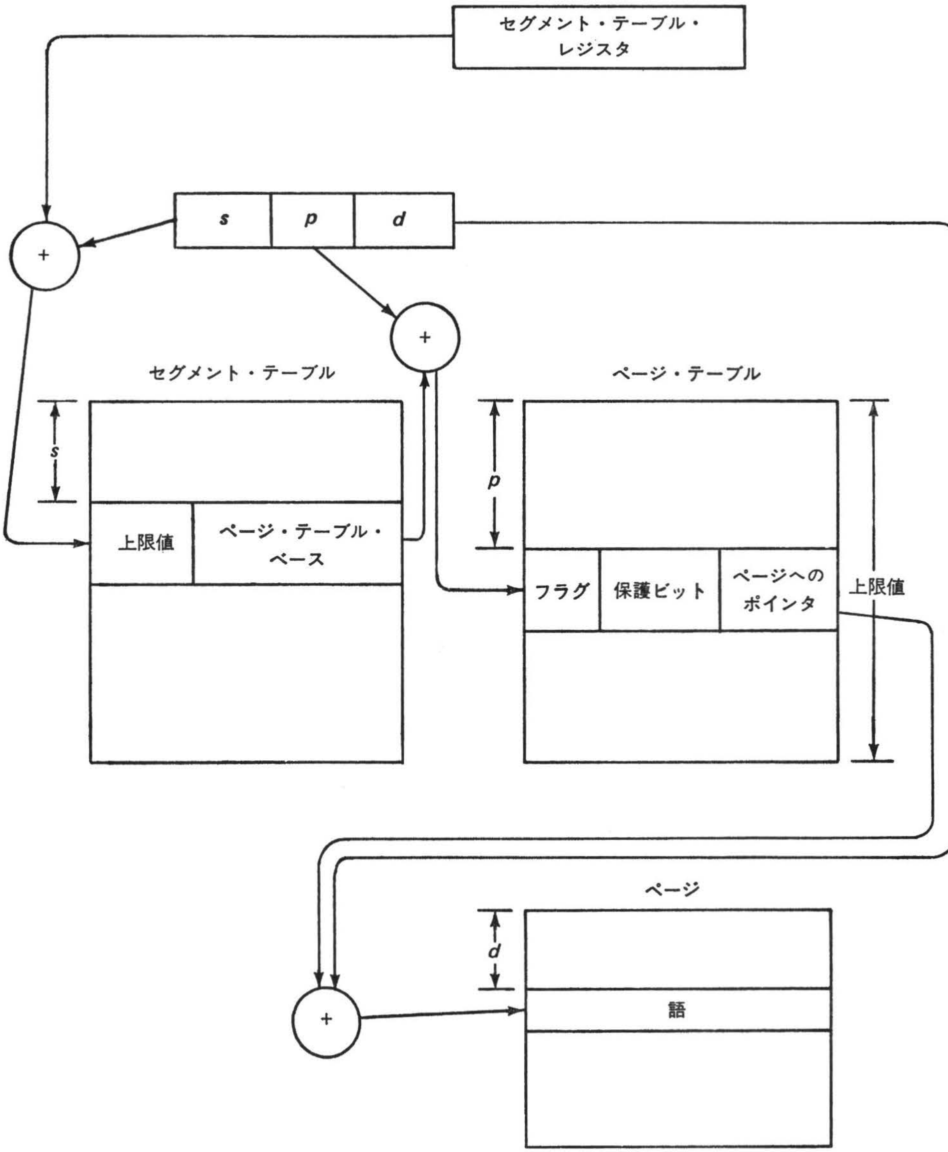


図 4-8 ページ化機構を持ったセグメント化における番地の計算. ［c(語)＝c(d ＋c(p＋c(s＋c(セグメント・テーブル・レジスタ)))).］

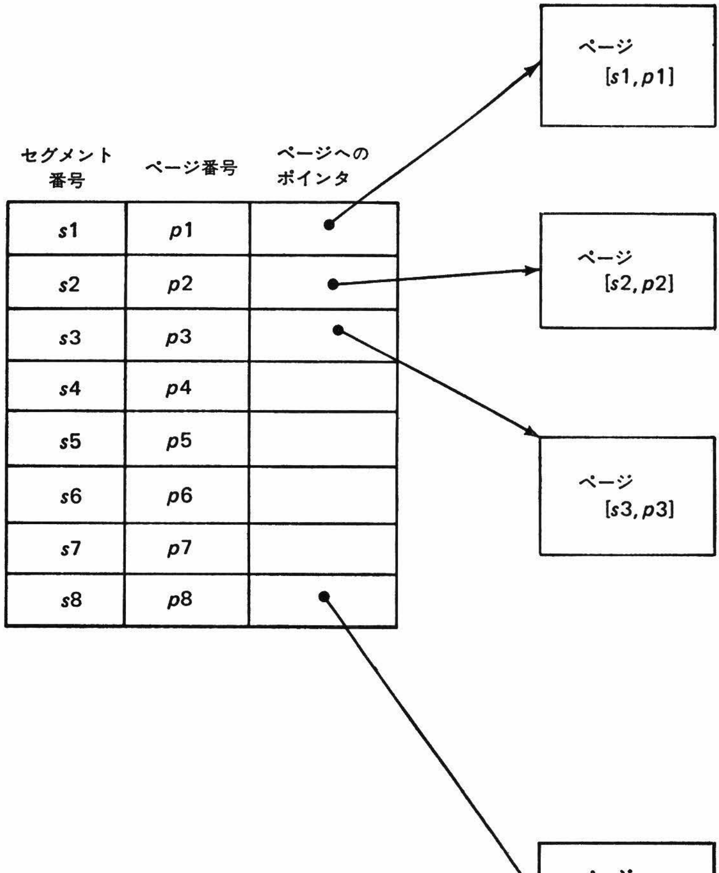


図 4-9 高速のページの番地付けに用いる連想レジスタ.（連想レジスタは，[*s*, *p*] からそのページの番地を得るために，一命令サイクルの間に，並列的に走査される.）

含まれていれば，そのページへのポインタを与えるので，必要な記憶サイクルを２つ減らすことができる．組み合せ [*s*, *p*] が連想レジスタに含まれていなければ，割り込みが発生し，スーパーバイザは連想レジスタの一つのエントリ [*s*, *p*] を新しく参照した [*s*, *p*] と置き換える．記憶の参照を観察してみる

と，少数の非常に頻繁にアクセスするページに集中する傾向がある† ので，ほとんどの番地参照は，必要な [s, p] の組み合せを連想レジスタの中に見つけることになる．

セグメント化-ページ化方式はセグメント化およびページ化の両者の利点をすべて持っている．プログラムの番地空間はセグメント化されているので，共有や結合は直接行なうことができる．記憶の番地空間はページ化されているので，記憶の割り当ては単純なセグメント化の方式以上に簡略化される．また外部断片化もなくなってしまう．このような柔軟性を持つためには，3要素への番地の参照が代償として必要になる．しかしながら，適当な連想ハードウェアを用いることにより，記憶の参照に必要な余分のコストは許容できる限界内に押さえることが可能となる．

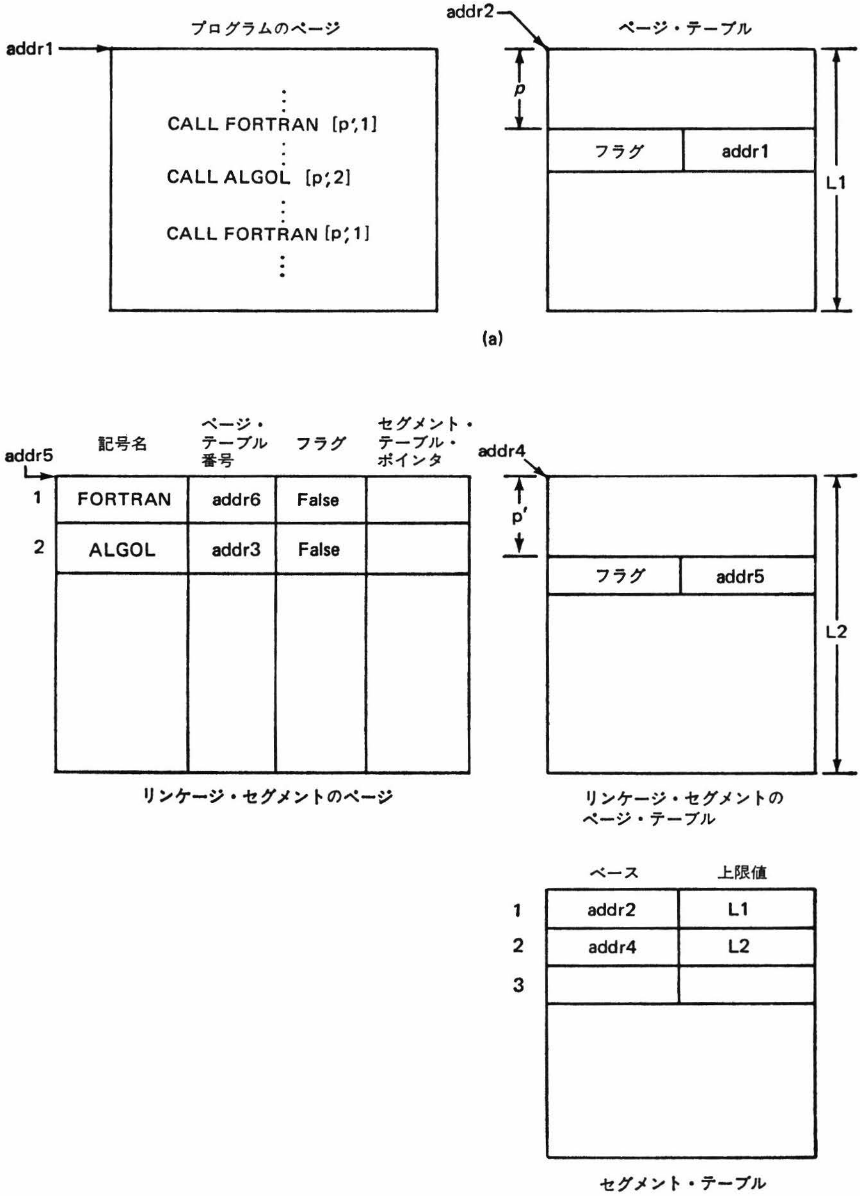
4.9 ページ化機構を持ったセグメント化を用いた結合

4.9.1 静 的 結 合

前節で注意したように，リエントラント型手続きは複数個のジョブによって共有できる．しかし，このような手続きの結合はしばしば非常にやっかいなことになる．ページ化機構を持ったセグメント化はこのような共有ブロックの管理に対して，便利な機能を与えてくれる．セグメント化-ページ化されたシステムにおいて，共有手続きPが自分自身のセグメント内に格納されている場合を考えよう．ユーザがPの使用を希望する（すなわち，Pをサブルーチンとして呼び出す）ならば，ユーザ・プログラム内でPを外部参照したことになり，ユーザ・プログラムに付随したセグメント・テーブルへエントリをつけ加え，Pのページ・テーブルへのポインタをそのエントリに代入する．Pに対する自分のデータ領域も，ユーザのセグメント・テーブルにつけ加える．このセグメント・テーブルの更新は，リンケージ・エディタによりロードの前に行なわれることが多い．しかしながら，次節に示すように，それは実行時まで遅らせることが可能となる．

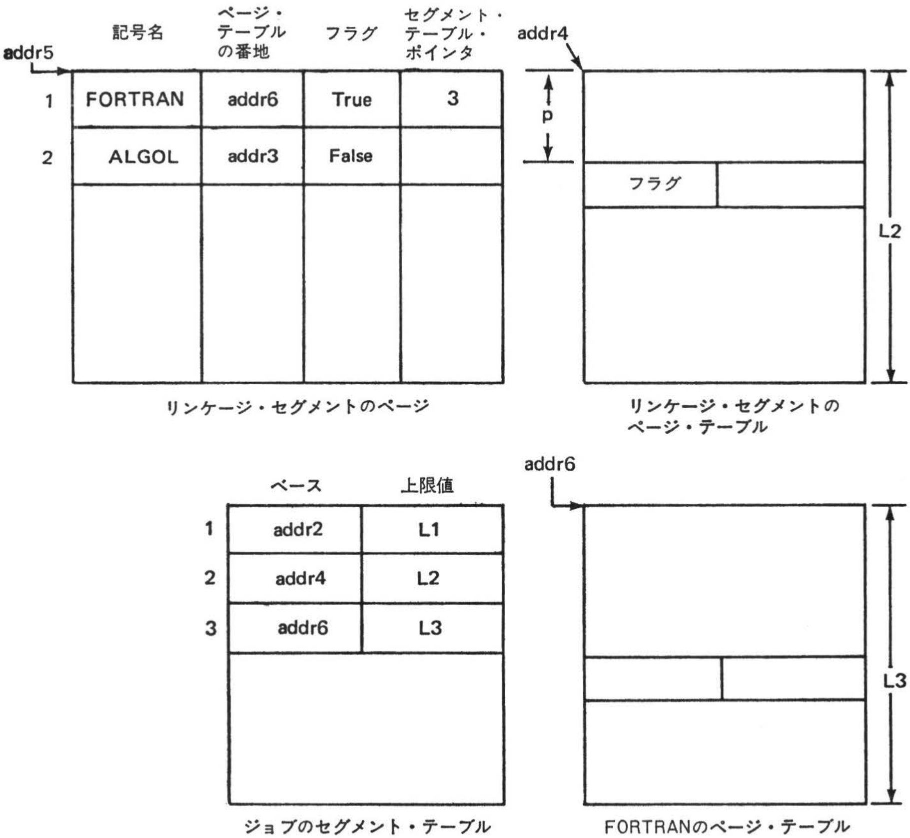
4.9.2 動 的 結 合

† 訳注． プログラムの局所性（locality）による（第 4.5 節参照．）



プログラムのページ
addr1
CALL FORTRAN [p',1]
CALL ALGOL [p',2]
CALL FORTRAN [p',1]
addr2
ページ・テーブル
p
フラグ
addr1
L1
(a)
addr5
記号名
ページ・テーブル番号
フラグ
セグメント・テーブル・ポインタ
1
FORTRAN
addr6
False
2
ALGOL
addr3
False
リンケージ・セグメントのページ
addr4
p'
フラグ
addr5
L2
リンケージ・セグメントのページ・テーブル
ベース
上限値
1
addr2
L1
2
addr4
L2
3
セグメント・テーブル
(b)

図 4-10a, b



(c)

図 4-10c 動的結合. (a) 共有セグメント FORTRAN および ALGOL を呼び出すプログラムのページとそのセグメントのページ・テーブル. (b) ジョブのセグメント・テーブル, リンケージ・セグメントのページ・テーブルおよび FORTRAN および ALGOL を結合する前のリンケージ・セグメントのページ. (c) ジョブのセグメント・テーブル, リンケージ・セメングトのページ・テーブル, リンケージ・セグメントのページおよび FORTRAN が結合された後の FORTRAN のページ・テーブル.

セグメント化を用いれば, 共有セグメントの結合を実行時まで遅らせることが可能となる. それゆえ, 実際に結合するモジュールはデータによって左右される. この節では, セグメント化-ページ化システムを用いた動的結合の一つ

の実現方法を論じてみよう（**図 4-10** 参照）.

コンパイルした時に，そのジョブの各オブジェクト・モジュール（すなわち，セグメント）と共に**リンケージ・セグメント**（linkage segment）と呼ばれる領域を記憶内に格納する．このリンケージ・セグメントはそのモジュールで参照する各手続きに付随した部分領域を持っている．与えられた参照手続きに対して，それに付随したリンケージ・セグメントの部分領域は次の情報を含んでいる：

1. 参照される手続きの記号名.
2. その手続きのページ・テーブルの場所.
3. そのルーチンがすでに結合されているかどうかを示すフラッグ（初期値は，**偽**にセットされている）.
4. ポインタ（実行中に，初めてそのルーチンが呼ばれたときにセットされる）.

手続きへの各外部参照は，リンケージ・セグメント内のその手続きに付随している部分領域への参照となる．ジョブAを現在実行していて，手続き FORTRAN を呼び出したと仮定しよう．呼び出しを実行すると，Aに対するリンケージ・セグメントを参照し FORTRAN に付けられたフラッグをチェックする．それが偽であれば，FORTRAN がまだAに結合されていないことを意味するので，ハードウェアによって割り込みが発生しその手続きを結合するようにオペレーティング・システムに伝える．オペレーティング・システムが起されると，Aのセグメント・テーブルに新しいエントリを作り FORTRAN のページ・テーブルの場所を代入する．リンケージ・セグメント内のポインタはAのセグメント・テーブル内の FORTRAN へのエントリを指すようにし，そのフラッグを真にセットする．その手続きに対する以後の呼び出しはそのリンケージ・セグメントへアクセスし，正しいセグメント・テーブルのエントリへつながるようになる．FORTRAN をAに結合するときの全操作を**図 4-10** に示す．

動的結合の一つの顕著な利点は，共有手続き（たとえば，前の例における FORTRAN）の場所を結合のための情報を変更せずに，いつでも変えてもよい

ということである．リンケージ・セグメントでは，FORTRAN のページ・テーブルの場所だけにしか興味がないが，それはこの方式では結合時には固定しておかなければならない．そのページは他のページ化されたルーチンと同様に，コアに入ってきたりコアから出ていったりしている．同様の方式は．ページ化を用いず，また共有する手続きのページ・テーブルをコア内に固定せずに行なう場合にも考えられるであろう．

静的および動的結合方法を組み合わせることも，また可能であることに注意しよう．たとえば，ユーザが定義するモジュールはロードする前に静的に結合してもよい．コンパイラやシステムのルーチンのような共有モジュールは，必要に応じて動的に結合してゆく．MULTICS システム［Organick, 1972］はセグメント化－ページ化の番地付け機構を用いているが，**図 4-10** に示したものと同様な動的結合が可能である．

演 習 問 題

4.1 一つのプロセスの実行中に，単一の手続きが主記憶の異った場所にオーバーレイされてもよい．この手続きが多数のプロセスに共有されているとき，このオーバーレイによって生ずる結合の問題点を述べよ．どうすれば，この問題を解決できるか．

4.2 動的オーバーレイに比べて，セグメント化はどんな利点を持っているか．ユーザがセグメント構造よりも，オーバーレイ構造の方を観迎するような状況があるであろうか．

4.3 Burroughs B 5500 のような純粋なセグメント化だけを用いる機械における記憶の割り当て方式を述べよ．

4.4 ある機械では記憶の語に番地付けができる．また，他の機械では語のほかに，記憶のバイトに番地付けできる．さらに，もっと別の機械では記憶の個々のビットまたは任意のビット列に対して番地付けできる．各番地付け方式に対する利点と欠点とを述べよ．

4.5 あるシステムにおいては，コアに存在しているときに変更を受けないよ

うなページは，それらを置き換えるとき，ページ化用ドラムに書き戻さない．そのような決定機構を実現するためのソフトウェアの技法について，概略を述べよ．

4.6 命令コードの中で指定するベース・レジスタに対して，相対的な番地付けをする機械が与えられたとする．また，絶対的な場所で実行する機械命令を生成するようなアセンブラも，与えられたとする．独立にコンパイルしたモジュールを，再配置可能なモードで一緒になって実行するような環境を**ブートストラップ**（boot strap）**法**で作り出すための段階を概説せよ．

4.7 第4．8節に述べたページ化－セグメント化の方式において，各セグメントのページ・テーブルをコアに常駐させておく必要があるか．もし必要でなければ，そのときの実現方法について概略を述べよ．セグメント・テーブルは，何故，コアに常駐しなければならないのか．

4.8 ページ化－セグメント化の環境において，共有手続きのページ・テーブルをコア内に固定する必要がないときの動的結合機構を設計せよ．そのような方式では，どんな利点があるか．

4.9 MULTICS の仮想記憶のハードウェアについて，詳細なレポートを作成せよ．MULTICS において，結合がどのように行なわれるか，説明せよ．

4.10 要素（i，j）を読んだり書いたりする確率が，$\left(\frac{1}{2}\right)$**max（i，j）で与えられるような性質を持つ大きなスパース行列（sparse matrix）を割り当てることを考えよう．次の各場合において，線形の記憶空間に対して，効率よく行列を割り当てる方法を見つけよ．必要な空間を最小にすることを試みよ：

1. 項目が，固定長で小さい場合，
2. 項目が，固定長で大きい場合，
3. 項目が，可変長の場合．

第 5 章

仮 想 記 憶

5.1 は じ め に

システムにおける大部分の情報は，主記憶に留めておくことはしない．むしろ，ドラムやディスクのような大容量の周辺装置に，それを格納しておく．しかしながら，アクティブなプロセスだけが，主記憶内に格納している情報に直接アクセスすることができる．したがって，プロセスは，その活動性を保持するために入出力を開始しなければならないと考えられている．

仮想記憶システムにおいては，各プロセスは，自分の情報はすべてコアの中に存在しているものと期待している．システムは，この期待に応えるために，プロセスのデータや命令の未使用の部分を，周辺装置に留めておく．そのプロセスが，コアに存在していない情報にアクセスしようとすると，スーパーバイザに警告を発する状態に自動的になる．スーパーバイザは，プロセスがアクセスできるように必要な情報を周辺装置からコアに転送する．プロセスの実行は，その転送の終了するまでスーパーバイザによって中断されているので，後者の動作はプロセスによくわかっていなければならない．転送が終了すると，プロセスは，実行を再開するので，それが“知る”限りにおいては，要求している情報は，いつでも，コアの中に存在していることになる．プロセスの情報が，物理的に一個以上の周辺装置に存在しているかどうかはプロセスにとってはわからない．

本章では，仮想記憶に対する割り当て方式の設計と解析について，調べてみる．本章は，３つの節に分ける．第 5. 2 節は，システム設計者が使用できる特殊なハードウェアおよび記憶構成について述べる．第 5. 3 節では，仮想記憶を

実現する2つの方法，すなわちセグメント化とページ化に対する割り当て方式について論ずる．これらの2つの方法における問題について，簡単な比較を行なう．第5.4節は，仮想記憶システムにおけるプログラムとシステムの振舞に対するいくつかのモデルを示す．さらに，割り当て方式の解析および局所的な方策とシステム全体の性能とのバランスについて考える．

5.2 仮想記憶に対するハードウェア装置

情報の格納および検索に用いるためのハードウェア装置が，たくさんある．すなわちレジスタ，コア，ドラム，固定ヘッド・ディスク，可動ヘッド・ディスク，磁気テープ，パンチ用紙テープ，カードなどである．これらの媒体については，読者は基本的な知識を持っているものと仮定しよう．さらに，仮想記憶に対して特に適しているハードウェア機構が存在する．特に，多重レベル記憶とページ化ドラムとは，従来のハードウェアの設計に比べて仮想記憶の経済的可能性を驚くほど向上させたのである．この節では，これらの2つの記憶構

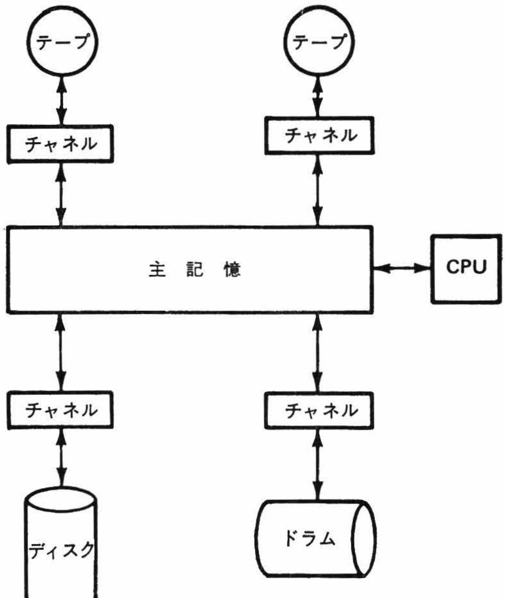


図 5-1　2レベル記憶の一般的構成

成の性質について調べてみよう．

5.2.1 多重レベル記憶

記憶構成の最も一般的なものは，2レベルから成る（**図 5-1** 参照）．レベル1は，コアのような大きな電子的記憶であり中央処理装置から直接番地付けができる．レベル2は，ドラム，ディスク，テープのような大容量の電子機械的な複数個の周辺装置から成り立つ．必要な情報がコアの中に存在していない（ページ化システムにおいては，このような状況はページ・フォールトに対応する）ときは，その情報を周辺装置からコアに転送してきて初めてアクセス可能となる．この方式は，初期の頃のページ化およびセグメント化の設計においてほとんど使われてきたが，現在でも，小型あるいは中型機の場合に多く見うけられる．しかし，命令の実行時間が100ナノ秒のオーダであるような大型高速機においては，基本的な2レベルの構成では不適当であることがわかる．

高速の CPU が，非常に遅いコアを用いて動いているときは，その大部分の時間は記憶からフェッチしたり記憶へ格納したりする操作を待つことに費されてしまう．大規模な**モノリシック**(monolithic)**記憶**は，近い将来，経済的に可能になるような徴候もいくつか見られるけれども，現在の技術のもとでは，非常に高速で大規模な**集積回路記憶**(integrated circuit memory)を用いるという解決法は経済的に不可能である.† そのような記憶は高いばかりでなく，高速の CPU が効率よく走るためには，一般に1～2メガ語 (million word) 必要となる．この問題に対する解決法は，多重レベルの主記憶を開発することであった [Kuck および Lawrie, 1970].

図 5-2 は N レベル記憶システムの形式的図式である [Mattson ほか, 1970]. 記憶装置 $M(N)$ は，電子機械的な装置である．他のすべての記憶のレベルは，異った速度を持つ主記憶である．一般に，$i>j$ ならば，$M(i)$ は $M(j)$ に比べて高速でかつ小規模であり高価（1語につき）なものである．以後の多重レ

---

† **訳注.** 大規模集積回路 (Large Scale Integration；略して **LSI**) 技術の発達に伴い，主記憶は，コアから半導体素子へ移行しつつある．記憶装置のハードウェア技術の進歩は急速であり，よくそれを理解していなくてはならない．ただし，主記憶に対するプログラムからの要求も増大する一方であり，ここで議論している多重レベルの記憶構成は，依然として重要な技法である．

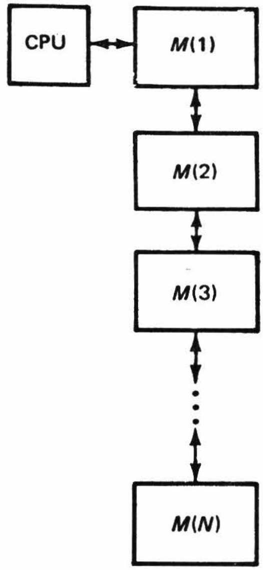


図 5-2 多重レベル記憶

ベル記憶の議論においては，記憶はページ化してある（すなわち，すべての転送は，固定長のブロックで行なう）と仮定する．

i レベルに格納されているページは，i 以上の番号のすべてのレベルに格納されている．もしも，CPUがレベル1にない語にアクセスしようとするならば，システムは必要とする語を含んでいるページを求めて大きな番号のレベルの中を捜し始める．そのページが見つかると，CPUはその語にアクセスすることが許され，そのページがレベル1に到達するまで高速の記憶のレベルへ転送してくる．ページを高い番号のレベルから低い番号（高速）のレベルへ転送する際には，古いページと置き換える．置き換えるページは，通常，最後のアクセスが最も古いものである．これは現在最も頻繁にアクセスしているページは，高速の方の記憶に存在すべきだという考えに基づいている．

この構成はいくつかの大規模な機械，たとえば，IBM 370/168 や IBM 370/195において実現されてきた［Liptay, 1968］．IBM の実現方法においては，レベル1［**キャッシュ**（cache）と呼ぶ］は100ナノ秒のシリコンのレジスタ記憶が

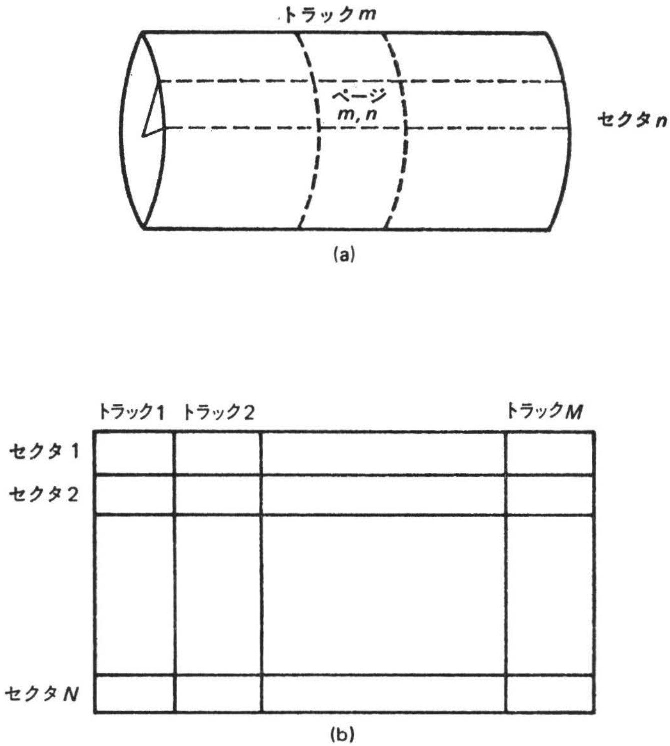


図 5-3　ページ化ドラム．(a) 物理的なドラム配置；(b) 平坦に展開したドラムの表面

16K～32K あり，レベル 2 は 1 マイクロ秒のコアが約 1 メガ語あり，レベル 3 すなわち外部レベルは，ページ化ドラムである．シリコンのレジスタ記憶とコアとのインターフェイスは，記憶のレベルがユーザに見えないようにするために，すべてハードウェアで実現している．3 レベルの記憶に対する平均アクセス時間は，レベル 1 のものに極めて近いという測定の結果を得ている．

5.2.2　ページ化ドラム

二次記憶から主記憶への転送率を高めるために，特殊なハードウェア装置が採用できる．そのような装置の一つである**ページ化ドラム**（paging drum）は，標準的なトラック単位のドラム構成に比べて非常に効率を上げることができる．

ページ化ドラムは，**図5-3**に示すように，論理的にはnセクタ（sector）とmトラック（track）とに分割されている．各論理的トラックに対して，読み書き用ヘッドの集合を与えてある．すなわち，一つの論理的トラックは複数個の物理的トラックになることがある．ドラムへの要求の待ちのリストは，セクタ番号毎にもうけてある．したがって，あわせてn個の待ちのリストが存在する．これらのリストをドラム制御装置に接続するときは，i番目のセクタが読み書きヘッドの下にちょうど来たときに，i番目の待ちのリストをドラムに接続するようにする．このような構成を用いれば，ドラムは次の転送を始めるために回転のある部分を待たなければならないという事態は，めったに起らないであろう．すなわち，ドラム・チャネルの使用率はきわめて高くなる．従来のように，セクタに分割せず，**先来先処理**（first-come first-served）の単一待ち行列により処理する設計方式は，セクタに分離し，多重リストを用いた設計方式に比べてずっと効率が悪い．ページ化ドラムの原則は，一つのトラックに一つのヘッドを持つような固定ヘッド・ディスクにも適用されてきた．ページ化ドラムの完全な解析は，Weingarten［1966］の文献を参照されたい．

5.3 セグメント化およびページ化における割り当て方式

概念的に考えると，セグメント化とページ化は，プログラムの番地空間および記憶の番地空間に適用される論理的および物理的な方式と考えられる．実際の実現場面においては，概念的な枠組は論理的なセグメントを物理的な記憶に格納する問題およびプログラムを固定長のページに分割する問題に変っている．したがって，実現する立場から見れば，ページ化とセグメント化の主要な差異は，単に前者が固定長のブロックを用いているのに対して，後者は可変長のブロックを用いていることである．したがって，ページ化とセグメント化は，非常に異った概念ではあるけれども，2つの方式を実現する問題点はよく似ており，それらを一緒に論ずる方が適当であろう．

セグメント化やページ化を実現する際には，3つの基本的な問題を考えなければならない：

1. 各レベルの記憶において，空間を，どのようにして割り当てるか．

2. ページやセグメントを，二次記憶から主記憶に転送する時期を決定するのには，どんな判定基準を用いたらよいか．

3. ページやセグメントを，二次記憶から主記憶に転送するときに，新しく転送してくるものに対して空きを作るために，置き換えるべき古いページやセグメントを決定するのにはどんな判定基準を用いたらよいか．

第一の問題は，ページ化においては実際は問題にはならない．記憶は固定長のページ・フレームに分割されており，システムはページとそれに対するページ・フレームの場所との表を管理している．空き空間の割り当ては，与えられた時間において，どのページ・フレームが使用されていないかを知る問題になってしまう．

可変長のセグメントを使用しているときは，この問題はずっと難しいものになる．記憶に対する要求があるとき，空間を割り当てるための妥当な方式がいくつか考えられる．第5.3.1節においては，これらの方式を紹介し比較している．

第二の問題，すなわち二次記憶から転送すべきページを選択することは，**フェッチ則**（fetch policy）または**プル則**（pull policy）と言われている．計算の中で実際に参照される前に二次記憶からページを転送する方式は，**前ページ化**（prepaging）という．参照したときにページ・フォールトになる時だけページを転送する方式を，**要求ページ化**(demand paging）という．前ページ化を用いるか，要求ページ化を用いるかという問題は，第5.3.2節で扱うことにする．

ある特定のページを，二次記憶から主記憶に転送しなければならないとき，新しいものに対して空きを作るためにどのページを置き換えなければならないか．多くの**置き換え則**（replacement policy）が提案されてきたが，そこには，実現することは不可能ではあるけれども最適であることを保証するものもある．第5.3.3節においては，いくつかの良く知られている置き換え則を紹介する．第5.4節では，プログラムやシステムの振舞のある原則に関して，これらの法則の評価を行なう．

5.3.1 可変長セグメントに対する割り当て方式

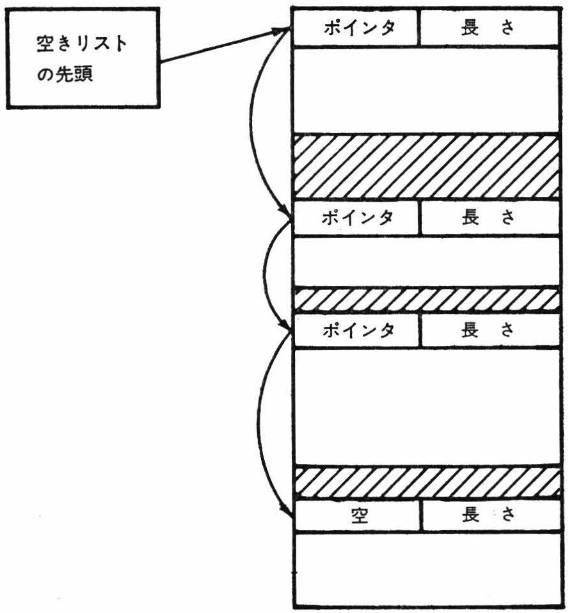


図 5-4 空きリストを用いた記憶管理（斜線部分は，空でない）.

主記憶は，K 語のブロック単位に割り当てられると仮定しよう．ここで，K は，典型的なものとして 8 ～2000の範囲にあるものとする．たとえば，すべての記憶のブロックが 1024 語であるようなシステムにおいて，100K 語を必要とするジョブは 100 個の連続したブロックを要求する．目的は，記憶の使用率が高く，割り当てアルゴリズムに対して必要なオーバーヘッドを少なくするような割り当て方式を見つけることである．

5.3.1.1 使用可能空間の結合リスト

記憶の管理をする最も簡単な方法は，空き記憶領域を番地の大きさの順に並べた EMPTY LIST と呼ぶ結合リストを用いる方法である．リスト内の各空き記憶領域は，次の領域への POINTER と，その領域に含まれるブロックの数を指定する LENGTH のフィールドとを含んでいる（**図 5-4** 参照）．以前に割り当てた領域が，その持ち主によって解放されたときは，EMPTY LIST へ容易に加えることができる．EMPTY LIST の要素を，順々に探索する．そし

て，新しく解放された領域を，番地の昇順を保つようにリスト内に併合する．もしも，EMPTY LIST における直前または直後（あるいは両者）の領域が，新しく解放された領域と連続しているならば，それを新しく解放された領域に併合して単一の空き領域を作ることができる．

EMPTY LIST から記憶を割り当てる方式には，**先適合**（first-fit）および**最適適合**（best-fit）の2つの方法が，よく知られている．長さがKブロックの領域に対する要求があるものと仮定しよう．両方の方式において，適当な大きさの領域が見つかったとき，最初のKブロックを要求したプロセスに対して確保し，残りの領域は EMPTY LIST に戻す．先適合方式においては，EMPTY LIST を先頭からたどり，LENGTH$\geq K$ が成り立つ最初の空き領域が使用される．最適適合方式においては，LENGTH$\geq K$ が成り立ち，その中で最小の領域を用いる．ちょうどKブロックの空き領域がない場合には，最適適合方式では各要求に対して EMPTY LIST 全体をたどらなければならない．したがって大部分の要求に対して，最適適合アルゴリズムは先適合の場合よりも遅くなる．またシミュレーションをいくつか行なってみると，大抵の場合，先適合の方が最適適合の場合よりも記憶を詰めることになるという結果が得られている [Knuth, 1968]． どちらの方法を行なうにしても，少なくともKの大きさの領域がないときには，その要求は失敗する．そのような緊急事態の基準については，次節で論ずる．

EMPTY LIST に替わるべき構成は，番地の昇順に並べないで LENGTH の昇順にリストを並べることである．このとき，先適合と最適適合は同一のアルゴリズムになる．このリストへのポインタ $\{P_1, P_2, \cdots, P_i, \cdots, P_m\}$ を持っていれば，さらに時間の節約をすることができる．ポインタP_iは，$i*c$から$(i+1)*c$ の大きさのブロックを持つ領域のリストの先頭を指している．ここにcは実現方法によって決まる定数である．そのようなポインタの集合は，探索を高速化することができるが，ポインタの内容の保守には費用がかかる．この方式の鍵となる問題は，新しく解放されたブロックを EMPTY LIST へ併合するときに，困難で時間がかかることである．

リストを用いた第三の方式は，**相棒システム**（buddy system）といわれるが，2**K の大きさのブロック単位に記憶領域を割り当てる方式である．記憶が

2**Sの大きさであれば，システムは（$S+1$）個のリストを保持するが，LIST（K）は 2**K の大きさのブロックのリストである．初期状態においては，記憶の第一語を指す LIST(S)以外すべてのリストは空である．2**K の大きさのブロックを割当りてるためには，LIST(K) の探索を始める．それが空でなければ，そのブロックのうち一つを割り当てる．そうでなければ，LIST ($K+1$) をチェックする．それが空でなければ，そのブロックのうちの一つを二等分し，一方を割り当て残りの一方を LIST(K) に加える．LIST($K+1$) が空であれば，空でないリストが見つかるまで，この過程を繰り返す．$K \leq J \leq S$ なるすべてのJに対し LIST(J) が空であれば，この要求を満足させることができない．

この割り当て方式を用いると，2**Kの大きさのブロックは，すべて2**Kの整数倍の番地から始まることが保証される．したがって，2**K の各ブロックに対し，それを作るために分割したブロック（長さ 2**($K+1$)）の番地を見つけることは容易である．2**($K+1$) の同一のブロックから分割された 2**K の大きさの2つの整合するブロック［すなわち“**相棒** (buddies)”］が解放されたときは，必ずそれらを併合する．相棒は，その最初の語の番地によって一意的に識別できるので，併合は簡単な操作である．シミュレーションによる実験では，この方式の方が先適合方式よりもいくらか速いことがわかっている．しかしながら，要求が1から2**S に一様に分布してるときは，相棒システムは，内部断片化のために先適合に比べて，その三分の一ほど余分に記憶の空間が必要となる［Knuth, 1968］．

5.3.1.2 詰　め

記憶が一杯になってくると，大きな記憶への要求を満足させることは，だんだん困難になってくる．最大の空き領域の大きさを越えるような領域に対する要求を受け取ったとき，3つの可能な処理の仕方がある．一つは，その要求を単に拒否することである．もう一つの方法は，現在，記憶内にあるセグメントを**先取り** (preempt) することである．すなわち，その記憶を取り上げて新しい要求に割り当ててしまう．第三の可能性は，いくつかの小さな領域を併合して，その要求に応えることができるような大きさの単一ブロックを得るように記憶を再構成することである．この最後の方法を**詰め** (compaction) という (図5-5参照)．

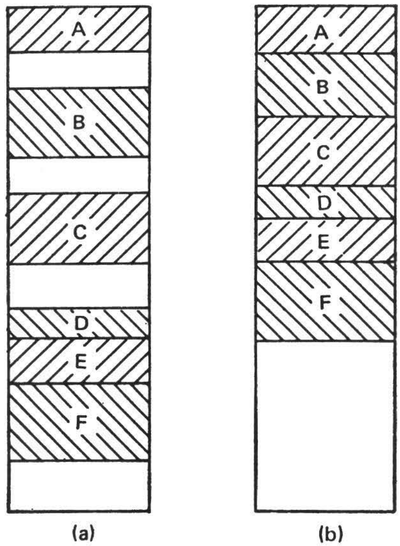


図 5-5 記憶を詰めること．(a)詰めの前；(b)詰めの後．

ばらばらの大きさの記憶に対する要求のあるシステムにおいては，先適合アルゴリズムを用いたとき，記憶がほとんど一杯になるまでは，詰めは一般には必要とならない．記憶空間は，どっちみち足りなくなるので，ほとんどの記憶を再配分するために必要な計算のオーバーヘッドは，通常，割りが合わない．しかし，記憶の要求が大きいときは，外部断片化のために記憶の三分の一が未使用であるような状況をすぐに想像することができるであろう（**図 4-3** 参照）．そのような状況においては，詰めは経済的にも可能なやり方である．

処理装置が，アクティブ・ジョブによってずっと使用中の状態に保たれていない場合は，詰めは，やはり利益をもたらすであろう．処理装置のアイドル時間は，今後のアクティブ・ジョブに空きを作るために，記憶を詰めることに使用することができる．新しいアクティブ・ジョブは，処理装置を働き続けさせるのに役に立つであろう．処理装置は，もしも，詰めの仕事に従事していなかったならば，アイドルになっていたのであるから，この場合の詰めの操作は無料であると考えられる．

Knuth は，使用可能な領域の数が平衡点で安定してしまうようなシステムに

おいて，先適合の割り当て方式を用いると，使用不可能な領域の数は近似的には使用可能な数の二倍になるという報告をしている．しかし，使用可能な領域は，小型化する傾向にあり断片化をもたらすので，われわれが期待するよりももっと深刻な問題となる．

動的な記憶割り当てに関してもっと知りたい読者は，Knuth［1968］の本を参照されたい．

5.3.2 フェッチ則

前節においては，記憶におけるセグメントの割り当てについて論じた．すべての記憶のブロックは同一の大きさなので，ページの割り当ては自明の操作である．次の問題は，二次記憶から転送すべき情報を，どうやって選択するかということである．ページ化の場合を最初に扱い，続いてセグメント化の場合に一般化してみよう．

二次記憶から主記憶へ転送するページを決定する際には，2つの場合を考えなくてはならない．すなわち，前ページ化か要求ページ化かである．近い将来，どのページが必要になるかということが見つけられ，プログラムの振舞が予想できるならば，明らかに前ページ化を用いた方が，ページ・フォールトを少なくすることができるであろう．ページ・フォールトを無くすことによって得られる節約量は，割り込みの処理のためのオーバーヘッドと，**前ページ**（prepage）すべきページを選択する処理のためのオーバーヘッドの差できまる．前ページ化しようとしまいと，新しいページは，主記憶の中に転送して来なければならないのであるから，ドラムからそのページを転送するのに必要な仕事は，節約の対象にはならない．もしも，あるページを，時期尚早にコア内に転送してきたならば，すなわち前ページ化しても決して参照されることがない場合には，不必要な操作によってある損失が発生することになる．この損失は，ページを選択するのに用いる処理装置の時間，コアに転送するのに用いるチャネルの時間，一時的にページからコアを取り上げるための費用などから成る．前の議論から考えると，時期尚早に，一ページを前ページ化することによって生ずる損失は，一ページを正しく前ページ化することによって得られる利益よりもはるかに大きい．したがって，前ページ化が損しないためには，前ページされたペ

ージを非常に高い確率で参照しなければならない．前ページ化のアルゴリズムとしては，前ページする情報が，後から転送してくるページによって重ね合せが起る前に，参照されることを確信できるようなものでなければならない．

残念ながら，プログラムの未来の参照を予測するのは，難しい．それゆえ，大抵の場合，要求ページ化の方がよい方式となる．実際，ある費用の仮定のもとでは，前ページ化は，要求ページ化に対して決して得にはならないという結果が示されている［Aho ほか，1971］．それでも，前ページ化の方法を用いたくなるような特殊な状況が存在する．たとえば，各プログラムは，その実行を開始する前に，システムの表をいくつかとその初期設定のための命令部分を必要とすることが知られているであろう．このような場合，ページ・フォールトを介してそれを要求するまで待つかわりに，すぐにそれらに対応するページをロードしてくるのは意味のあることである［Oppenheimer および Weizer，1968］．セグメント化－ページ化方式においては，同様の考え方が適用できる．新しいセグメントに入ってきたときには，そのうちのいくつかのページは，非常に高い確率で参照されるであろう．この場合も，前ページ化は，ページ・フォールトを減少させる．しかしながら，大部分の状況においては，前ページ化は損失の多い方式であり，誤った推測による損失は正しい推測によって得られる利益をはかるかに越えてしまう．

非要求ページ化は，稀にしか見られない．非要求型のセグメント化も，やはり稀である．前ページ化は，ページの間のある論理的な結合がわかっているときに，主として役に立つ．しかし，大抵のセグメントは，すでに完全な論理的な単位であるから，参照の振舞を予測する気はあまり起らない．したがって，セグメントのフェッチは，ほとんどの場合，要求に基づいて行なわなければならない．以後のこの章では，ページ化とセグメント化は，要求に基づいて行なうものと仮定する．

5.3.3 ページ置き換え則

あるページをコアの中に転送しなければならなくなったとき，問題は，それをどこに置くかということである．未使用のページ・フレームがあるときには，何の問題もない．しかし，大抵の場合，すべてのページ・フレームにはページ

が存在しているので，どのページと置き換えるかということを決定しなければならない．**置き換え則**（replacement rule）は，そのような決定をするための方策である．これらの規則は，セグメントとページの置き換え両方に対して同じように適用することができるけれども，統一をとるために，ページ化に関する言葉を用いて以後の議論を進めてゆこう．

置き換え則は，将来のページの移動量の期待値を，どの位最小化できるかという点から判断する．考え方の基本は，“使用できる”ページは，置き換えないということである．それゆえ，参照することが期待できるようなページよりも，参照することが期待できないようなページを置き換えるものとして選んだ方が一般によいことになる．単一の規則あるいはいくつかの規則を相互に組み合わせたものから成るある与えられた方式を，局所的にまたは全域的に適用することができる．置き換え則を局所的に使ったときは，記憶を作業空間に分割し一つの作業空間に一つの実行ジョブが存在するようにする．各作業空間に対しては，他の作業空間とは全く無関係に決められた規則を適用する．全域的に適用するときには，記憶全体を一つの単位とし，どのジョブが要求してきたかということとは無関係に記憶空間全体に置き換え則を適用する．

この節では，より一般的な置き換え則をいくつか論ずる．これらの規則の振舞については，章の終りで解析する．

5.3.3.1 ランダム選択

この方式では，消去されるページをランダムに選択する．勿論，この方式の実現は，非常に簡単である．しかし，あきらかに，使用できるページも頻繁に除去されてしまう．

5.3.3.2 先入先出（FIFO）

ここでは，“最も古い”ページがいなくなる．ただし，最も古いとは，コアの中にいた時間が最も長いことをいう．FIFO は，各プロセスの記憶領域の中で局所的に用いるか，コア内のすべてのページに対して全域的に用いることができる．これを実現するのは，まったく簡単である．記憶の中のページ・フレーム（それらがK個あると仮定しよう）は，最も新しいページへのポインタを持ったK個の要素の連続したリストを持っている．ページが転送されてくるときには，そのポインタを１つ増加（モジュールKで計算）して，そのとき指し

ているページと置き換える.

この手続きの欠点は，システムが，中位からきわめてこんできたときに，使用可能なページがたくさん除去されてしまうので，非常にまずく働くということである．あきらかに，最も古いページが，最も使用しないページになるとは限らない．さらに，プログラムにおいては，命令をフェッチするパターンは，順次的であるというはっきりした実験結果はあるけれども，データへの参照は，すべて順次的に行なわれるというわけではない．もし，そうであれば，この方式は理想的なものである．実現の簡潔さは魅力的であるが，FIFO は重大な性能上の限界を持っている.

5.3.3.3　下駄ばき先入先出 (Biased First-In-First-Out；略して **BIFO**)

これは，局所的に適用する方式である．アクティブな各ジョブ i には，$n(i)$ ページの集合を割り当てる．ただし，一般に $i \neq j$ に対し，$n(i) \neq n(j)$ である．$n(i)$ の集合の中では FIFO が適用される．t 時間単位毎にすべての i に対し $n(i)$ を変更するので，ユーザの用いる集合は増加したり縮少したりする．したがって，ある時間においては幾人かのユーザを優遇している．$n(i)$ を変更することにより，優遇されるユーザは全体に分布する．パラメータ $n(i)$ と t とは，システムおよびジョブの性質によって選択する.

5.3.3.4　最長未使用 (least recently used；略して **LRU**)

この方式では最も長い間参照されなかったページを除去する．LRU の置き換え則は，実現するのがかなり難しい．一番良く用いられているソフトウェアの実現法は，各ページに与えた1ビットを用いる方法である．コア内にある各ページに対して，そのページへの最後の参照からの時間間隔の数を示すエントリを持ったテーブルをシステムが管理している．あるページが参照されると，そのページのビットがセットされる．一定の時間間隔ごとに，すべてのビットを見てどのページがその間隔の間，参照されていなかったかを調べる．そして，これらのページの番号をテーブルに記録する．すなわち，これらのページに与えた非参照のカウンタを，それぞれ1つづつふやす．新しいページを転送するときは，このテーブルを参照する．最も長い時間間隔の間，参照されなかったページが置き換えられる.

---

† Belady および Kuehner [1969].

経験的には，LRU はほとんどの場合うまく働くことが知られている．この方式の主なる欠点は，実現のための費用がかかることである．ハードウェアを用いると，この規則を経済的にも可能とすることができるだろう．たとえば，ハードウェアによってページを参照するたびに，参照ビットをセットすることができる．多重レベルの記憶においては，ページの置き換えは，普通ハードウェアで実現したLRU を用いる．結局のところ，LRU はページの置き換えに対する最も一般的に用いられる方式となるであろう．

5.3.3.5 事前に決めた優先度

この方法では，プログラマやコンパイラが，どのページが近い将来高い優先度を持つべきかを，特別の命令を用いてシステムに伝える．もしもプログラミング言語が高度に構造化されていれば，ページ置き換えの決定を助けるために参照パターンを検出することができる．事前に決めた優先度を用いて，システムは近い将来，どのページが最も参照されないかということを判定することができる．

5.3.3.6 システムが決める優先度

同様にして，置き換えアルゴリズムはスケジューリングのために各ジョブに割り当てている優先度と関係づけることができる．現在，システムの中に存在するジョブは，待ち行列の中でそれらの優先度に従って並べられている．ページ・フォールトが起ると，待ち行列を走査して，コアの中にページがあり最低の優先度を持つジョブを見つけ出す．もし，同じ優先度であれば，他の規則たとえばシステムに存在していた時間などによって順番をつける．システムは，このジョブのページからたとえば LRU の規則などによって置き換えるものを一つ選ぶ．ページ・フォールトを起したジョブが，最低の優先度を持っているならば自分自身のページの一つと置き換えなければならない．そのジョブの最後のページがスワップ・アウトされてしまったならば，他のジョブが終了してページ・フレームをいくつか解放するまで待たなければならない．

このアルゴリズムの欠点は，最高の優先度を持つジョブが一度参照したけれどそれ以後一度も参照しないようなページを多数たくわえてしまうことである．他のプログラムはページ・フレームをそれから取り上げることは許されていないので，このような役に立たないページがプログラムの実行が終了するまでコ

アの中に残ってしまう.

5.3.3.7 最適置き換え

最後に, Belady [1966] によって提案され Mattson 等 [1970] によって最適であると証明されたOPTというアルゴリズムを紹介する. この方式は, 参照の頻度に対して最小のページ・フォールトを発生する. OPT は, 次の参照が一番遠い将来起るようなページ, すなわち一番長い期間参照することがないようなページを置き換える方式である. これは, 最適のアルゴリズムではあるけれども一つの重大な欠陥を持っている. すなわち, ある与えられた時間において, 将来の参照に対して誰も完全にわかることはできないので, これを実現することは不可能なのである. しかしながら, OPT は他の実現可能なアルゴリズムに対して有益な比較の標準として用いることができる.

5.3.3.8 複合方式

前述した方法は, 簡潔でうまい定義を与えられていたが, 多くの複合規則あるいは常識的な規則も, 同様にうまく働く. たとえば, Spectra 70/46 の時分割オペレーティング・システム (RCA Corp.) では, 適合度の降順に並べたページの置き換えの選択のリストを用意している [Oppenheimer および Weizer, 1968]:

1. アクティブでないタスクの読み出しのみのページ,
2. アクティブでないタスクの書き込みされたページ,
3. いままでの半秒間に, アクセスされなかった制御プログラムのページ,
4. 入出力を待っているアクティブなタスクのページ,
5. 他のアクティブなタスクのページ.

ページのリストは, 各サブ・グループ毎に管理している. 置き換えが必要になったときは, 最も適合度の高い空でないリストの中からページを選択する.

一般に, ページの置き換え方式は設計の際に選択が難しい問題である. LRU のようにうまく働くことがわかっているようないくつかの方法は, ハードウェアの支援なしには実現が困難である. しかし, 多くの単純なアルゴリズムでは, 使用可能なページを多数除去してしまう. 次節では, 置き換え則の比較に用いる解析的技法について論ずるが, それが設計者が賢明な選択をするときの助け

になることを望んでいる.

5.4 ページ化システムの解析

これまでの議論においては，ページのフェッチおよび置き換えに対する規則について紹介してきた．システム設計におけるこれらの選択枝の中から妥当なものを選び出すためには，ページ化におけるプログラムおよびシステムの振舞をよく理解する必要がある．ワーキング・セット [Denning, 1968] は，このような振舞をモデル化する一つの試みである.

5.4.1 ワーキング・セット・モデルと局所性

ページ化した仮想記憶の環境においては，ある瞬間において一つのプログラムは，一般にそのページの集合の一部分だけをコアの中に持っているだけである．これらのコアに存在しているページが，頻繁に参照するものであるならばページ・フォールトは多くは起らずページの移動量は少なくなるであろう．簡単に言えば，あるプロセスのワーキング・セットとは，ページ・フォールトが起る前に“妥当な”量の処理装置の時間を得るためにコアの中に存在していなければならないようなページの集合のことを指す [Denning, 1968 ; Denning および Schwartz, 1972]．ページの参照パターンを観察して，どのページがワーキング・セットに属するのかを決めるのはシステムの責任である.

時刻 t におけるプロセスのワーキング・セット(working set) $W(t, T)$ は，そのプロセスの時間間隔 $(t-T, t)$ の間に参照したページの集合と定義する．ここで，プロセスの時間とは処理装置がそのプロセスに割り当てられている時間を指す．それゆえ，ワーキング・セットは，正確にはプロセスの時間の最後の T 秒間に参照されたページの集合を指すことになる.

ページ・フォールトを起すプロセスの命令の割合を e とし，コアに存在しているプロセスのページの割合を p とする．プロセス内のすべての番地の参照がランダム（すなわち，各参照は等確率ですべてのページに対して行なわれる）と仮定するならば，$e=1-p$ が成り立つ．しかし，プログラムはランダムでない参照パターンを持っていることが知られている．参照は，全体のプログラムの部分集合，すなわち現在アクセスしている命令やデータを含んでいる部分集

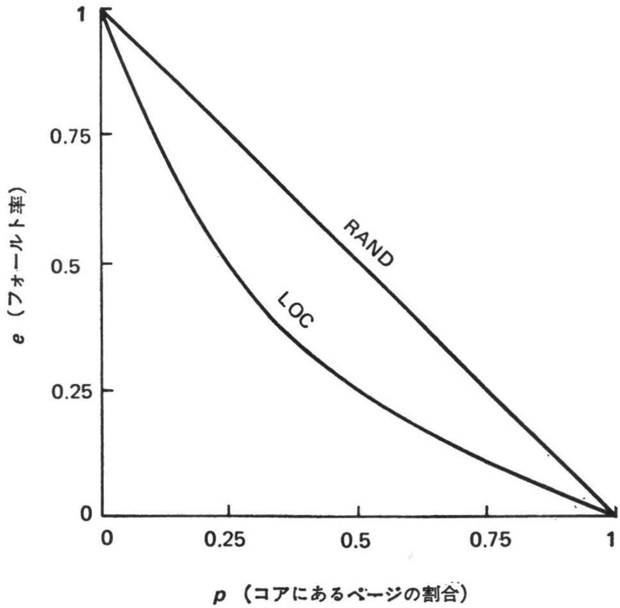


図 5-6 フォールトの割合に対する参照パターンの効果，(RAND：ランダムな参照パターン．LOC：局所性を持つプログラムの参照パターン)

合の近辺に集中する傾向がある．プログラムのこの性質，すなわち参照がグループ化されるという傾向を**局所性**（locality）と呼ぶ．プログラムの局所性を考えると，プログラムのページのうまい部分集合（たとえばプログラムのワーキング・セット）がコアの中にあれば，コアの中に余分なページをつけ加えても，フォールトの割合 e を本質的に向上させることはほとんどないであろう．

図 5-6 では，ランダムな場合と，集中的な場合との参照パターンが，フォールトの割合に対してもつ効果を比較している．LOC の曲線上では，プロセスのページの約三分の一がコアに存在すれば，それ以後 p を増加しても e を思ったように減らすことはできない．

このグラフにおいては，鍵となる仮定が存在する．各 p に対し p の割合の“最良”のページがコアに存在しているものと仮定している．さらに，プログラムの実行に際しては，システムは一番頻繁に参照するページが常にコアの中に存在するように，ページ・フォールトを処理しているものと仮定する．これ

らの仮定は，あきらかに用いている置き換え則の関数となる．たとえば，OPT はこれらの仮定を満すのに対して LRU は満さない．

Denning は LOC 曲線を最適化するのを助けるために，ワーキング・セットに基づいた置き換え則を提案した．しかし，ページ化の性能を改良するためには他の方法もいくつかある．ページの内部で，プログラムやデータを注意深く配置することによりページの参照が集中化しプロセスの局所性を増すことができる．これらの方法については，第 5.4.3 節において，いくつか述べる．

5.4.2 ページ化の振舞における異常事態

5.4.2.1 スラッシング

記憶または一つのジョブの記憶における作業空間が過重負担になると，システムの性能がページの移動量の増加のために著しく低下することがある．この現象を**スラッシング** (thrashing) という．

置き換え則が各ジョブの作業空間内で働くような局所的方式においては，もしもジョブに充分なページ・フレームを与えていなければ，各ページ・フォールトによって，有用なページが置き換えられることを余儀なくされるであろう．番地参照の大部分でページ・フォールトが起り，ページの移動量が危機的に高くなってしまう．全域的な方式の場合も，同様の結果を生む．記憶への要求が非常に高ければ，各ページ・フォールトは恐らく他のジョブに属している有用なページを置き換えることを余儀なくされるであろう．ページの移動量は増大し，そのために性能は低下する．その結果，大部分の計算時間はページ・フォールトの処理のために費される．そして，機械が全速力で働いても有益な計算は行き詰まったままになってしまう．

仮想記憶は，普通，各アクティブなジョブの三分の一がコア内に存在していれば，すばらしい結果をもたらすけれども，実記憶に対してはとってかわるものは何もないということを思い出さなければならない．大規模なプログラムは，どのような記憶管理システムの下であっても大量のコアを必要とするのである．

5.4.2.2 Belady の異常事態

ある与えられたページ化方式の効率を決定する際に，ある客観的な評価の手段を持つことは本質的に重要である．常識や直観は，非常に多くの場合，誤っ

た結果につながる．たとえばプログラムに割り当てられるページ・フレームの数が増加すると，ページ・フォールトの起る回数は減少すると仮定するのが普通であろう．平均的には，これはたしかに正しいのであるが，全く正反対のことが生ずる場合がときどきある [Belady ほか，1969]．各ページが1から5までの番号の名前がつけられている5ページのプログラムを考えよう．FIFO を用いた場合，参照パターンが 1-2-3-4-1-2-5-1-2-3-4-5 である5ページのプログラムは4個のページ・フレームを与えられたとき10回のページ・フォールトを起すのに対して，3個のページ・フレームを与えられたときは9回しか起さないのである！

異常事態は，FIFO 置き換え則に一意的にあらわれるように見えるけれども，これは完全に人工的に作ったというわけではない．実際のプログラムがこのように動いたということが見つかっている．解析を注意深く行なわないと，驚くべき，また好ましくない結果につながることがあるという警告になっているのである．

5.4.3 プログラムの設計におけるページ化の影響

良い局所性を持っていると，低いページ・フォールトに押さえるために必要なページの数が減るということを前に述べた．プログラムを注意深く構成すれば，局所性を驚くほど改善できる．

5.4.3.1 ページづけ

プログラムを，ページに分割する方法を**ページづけ** (pagination) というが，これは局所性に対して重要な影響を持っている．プログラムのモジュールの配置を変更すると，ページ化の振舞に対して驚くべき効果が得られるという実験が行なわれてきた [Comeau, 1967]．プログラムを作る様式(style) も，ある与えられたページづけの方式の性能に影響を与えることが知られている．構造的プログラミング (structured programming)† の分野における最近の研究では，様式の一様化を提案している [Weinberg, 1971]．良いプログラミングの様式を明確に定義すれば，参照のパターンを翻訳時に知ることができるのでページ

---

† **訳注**．正しいプログラムを作るためにプログラミングで用いる制御構造に制限を加え，プログラムをなるべく構造的に作成しようという技法 (9.3節参照)．

づけをずっと効率良く行なえるようなコンパイラを構成することも可能となるであろう．しかし，効率が良く自動的なページづけをする方法を開発することは，未だ初期の段階にすぎない．

5.4.3.2 ATLAS におけるページの最適化

初期の要求ページ化システムの例である ATLAS システムでは，プログラムの各ループは同じページ内に含まれるようにした．この方式の目標は同じページ（またはページ群）を何回も参照するようにするということである．これは基本的にはうまいページづけを用いて，局所性の最適化をはかろうとする方法である．ループに対しては極めて効率がよいけれども，もっとランダムな参照パターンに対しては，これはうまく働かない．さらに，実現するために非常に費用がかかる可能性がある．

5.4.3.3 行列の最適化

ページ化システムにおいて，大規模行列を処理するような計算は，参照の局所性が少ないために性能の重大な低下を来たす可能性がある．McKellar と Coffman[1969] とは，行列演算を注意深く順序づけるとページ・フォールトを急激に減少させることができることを示した．しかしながら配列要素のインデックス化が複雑であるので，番地計算が少々高いものにつく．それにもかかわらず，フォールトの割合を減少したことによる利益の方が要素の添字を計算するために余計にかかる損失よりもずっと大きいように思える．また，大規模な連立方程式を**部分枢軸**（partial pivot）用いた前向き消去法（すなわち，Gauss の消去法）で解いたとき，それらを部分行列の集団として格納することにより大規模な行列に対して，フォールトの割合を大幅に改善することができる．

プログラムの振舞を，このように解析することはページ化に対するコンパイラを設計する際に役に立つ．これらの技法を適用することによりページ化の環境において，プログラムの性能を最適化するように言語翻訳システムやサブルーチン・パッケージを設計することができるであろう．

5.4.4 ページ化システムの設計における問題点

ページ化した計算機システムにおいては，よい設計をするために調整すべき独立のパラメータが幾つか存在する．そのうちの二，三をあげると，置き換え

則，コアの大きさ，ジョブの分割の構成，スケジューリングの方式，ページの大きさなどがある．この節では，システムの設計の決定に関する解析的な結果をいくつか論ずる．これらは，ページ化システムの解析に関して発表された論文をいくつか集めたものにすぎない．

5.4.4.1 置き換え則の比較

第5.3.3節で示した置き換え則の中では，OPT [Belady, 1966 ; Mattson ほか, 1970] が最適であることがわかっている．実際のシステムにおいては実現不可能ではあるけれども，OPTはシミュレーションすることができる．置き換え則のシミュレーションを行なうためには，一般にいくつかのテスト・プログラムを実行してその実行中に生ずる新しいページの参照を記録することから行なう．このページの参照の系列はシミュレーションへの入力として用いることができ，いろいろの置き換え方式と記憶の大きさとに対してフォールトの振舞いを決定する．今度は，将来の参照に関して完全にわかっているので，OPTが実現できることに注意しよう．これは他の規則との比較の基準を与えてくれる．

Coffman と Varian [1968] は，参照列の資料を作るために4つのテスト・

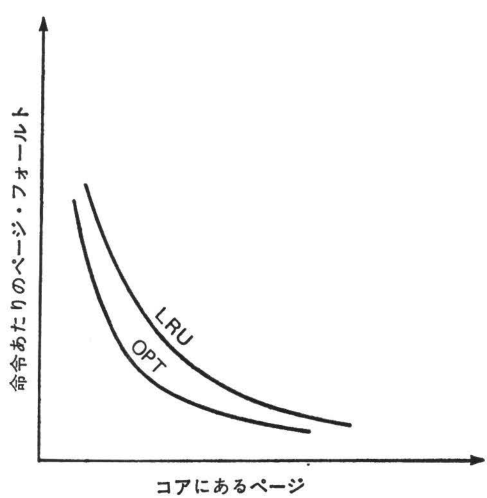


図 5-7 LRU 対 OPT 置き換え則

プログラムを選んだ．すなわち，SNOBOL のコンパイラ，WATFOR† のコンパイラ，フーリエ変換のプログラムおよび微分方程式を解くシステムである．これらのプログラムを，LRU と OPT の下でテストした結果が**図 5-7** に要約してある．すべての場合において，LRU と OPT の間の差は 30〜40 パーセントの間におさまることがわかった．Coffman と Varian は，コア内のページ・フレームを減少させると，ページ・フォールトが予想通り増加することを観察すると共に，コア内のページの数を増加させると，ページの大きさをふやすよりもずっと有用であることを見つけ出した（第 5.4.4.2 節参照)．

Belady と Kuehner [1969] は，FIFO と BIFO とのフォールトの割合を比較している．妥当な参照の局所性を示すようなプログラムでは，ページ・フレームの大きさに対して，ページ・フォールトの間の時間 [**フォールト間の時間**

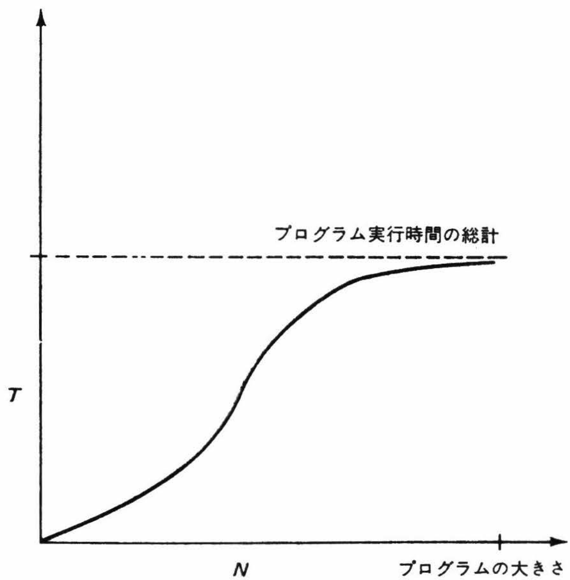


図 5-8 作業空間の大きさに対するフォールトの間の時間．(T，ページフォールトの間の時間；N，コア内のページの数.)

† 訳注．カナダの University of Waterloo で開発された学生実習用 FORTRAN コンパイラ．大量の FORTRAN ジョブが効率よく処理できる．

(interfault time) と呼ぶ] が図5-8に示すように分布している. 大部分のページ化方式では, 各ジョブに対して固定した量のページ, たとえばAページを与えている. Belady と Kuehner は, 数学的モデルを用いて, 次のことを見出した. すなわち, ジョブの半分の時間を $A-k$ ページ・フレームを用いて, また半分の時間を $A+k$ ページ・フレーム (ただし, k は, ある小さい整数定数) を用いて実行すると, フォールト間の時間の平均値が長くなる. このような下駄をはかせた方式の代表としてBIFO を用いると, BIFO は, 下駄をはかせない FIFO 則に対して, フォールトの間の時間について10〜15%の利得が得られることが実験的に示された.

前述の例では, シミュレーションと数学的解析とを用いてページの置き換え則の比較を行なっていた. しかし, 多くのシステムでは, 解析するのが難しいそのためだけに考え出した規則 (たとえば, Spectra 70/46) を用いている. また, 各システムの微妙な差が, シミュレーションの結果を一般的に適用することを困難なものとしている. それゆえ, 置き換え則についてはいろいろの研究が存在しているのにもかかわらず, どの規則が実現するのに "ふさわしい" ものであるかという意見の一致がシステムの設計者の間では見られない.

5.4.4.2 ページの大きさの決定

ページの大きさを最適化するときに考えなければならない要因がいくつかある. 内部断片化を少なくするためには, ページの大きさは小さくすべきである. しかし, ページの大きさが減少するにつれて, 大規模なプログラムの実行に必要なページの数は増大してしまう. その結果, 各プログラムのページ・テーブルの長さは増加する. 内部断片化とページ・テーブルの長さのみを考えるならば, 平均的な大きさのプログラムに対してページの大きさは約45語にすべきであるといわれている [Denning, 1970]. 残念ながら, 小さなページにするとページの移動量がふえ, ものごとをきわめて複雑なものにしてしまう. ページを二次記憶からフェッチしてくるときに重要なことはシーク時間であるから, ページの転送の回数が増加するとシステムの性能を非常に低下させてしまう. それゆえ, ページの移動量を低く押さえるためには比較的大きなページを用いるべきである. ページの移動量に対して, 断片化とページ・テーブルの長さとの釣合いをとるためには, ページの大きさは256〜1024語位のものが代表的なも

のと考えられる．

5.4.4.3 そのほかの結果

ページ化に関するいくつかの面白い問題が，形式化され解決されてきた．Coffman と Randell [1971] は，あるクラスの置き換えアルゴリズムに対して余分の記憶がつけ加わったとしたときに，システムのフォールトの割合を予測できることを示した．その予測に対して必要な情報はページの置き換えの記録だけであるが，それは記憶が拡張される前のシステムで，資料をとるための実行をすれば容易に得ることができる．このようにすれば，追加の記憶を実際に購入する前にシステムの振舞の改善を確かめることができる．

Denning [1968] は記憶と処理装置とへの要求が釣合うようなモデルを提唱した．処理装置と記憶への要求の予測を数学的関数を用いて行なうことにより，2つの資源の釣合いを保つようなジョブを選択することができる．この手法は資源の使用率を最大にするためには役に立つが，個々のユーザの待ち時間を減少させる方向とは一致していないかもしれない．

Belady と Kuehner [1969] は，大規模なジョブに対して空間の割り当てを制限したときの効果について研究した．彼等の結果によれば，コアの割り当てを制限するような割り当て方式を用いると，他のプロセスがその記憶を長時間使用できなくなるためにシステムの負荷が実際には増大してしまう．彼等は，膨大な記憶を必要とするようなプロセスは，その記憶を与えてなるべく早くシステムから抜け出させてしまうべきであると結論を出している．

Coffman と Ryan [1972] は，固定および動的に変化する**分割** (partition) の大きさについて追及した．彼等の結果によれば，大きさがあまり変化しないワーキング・セットに対しては固定長の分割がよさそうに見える．また時間と共に大幅に大きさの変るワーキング・セットに対して，動的な分割を用いるときに得られる節約の量についても定量化をした．

Rice と van Dam [1971] は，テキスト・エディタ・システムという特別な場合に対してページ化のパラメータの選択の問題を追及した．特にページの大きさ，置き換え則，ページづけ，ワーキング・セットの大きさなどについて考慮している．ページ化とセグメント化との比較も行なっている．彼等の結論によれば，テキスト・テディタでは両者の方法がうまく働くように見え，また方式

を選択する場合，実現の容易さが決定の要因になると考えられると述べている．

ページ化システムの解析については，過剰なほどの材料がそろっている．これは，一つには仮想記憶の重要性が増大したことにより，また一つにはページ化の問題が形式化し易いということによっているのであろう．これらの解析的手法の追及に興味のある読者は，注釈付き参考文献一覧表を利用して欲しい．

5.5 おわりに

本章の話題は，仮想記憶システムにおけるページ化の問題に片寄っているけれども，セグメント化システムだけを用いて非常に成功した例(特に Burroughs システム）もある．しかし，一般に可変長のセグメントの転送の実現は容易ではない．それでもプログラムやデータを可変長のセグメントに自然に断片化することができるので，この手法は直観的には魅力がある．あきらかに両者をとり入れたシステム，すなわちページ化－セグメント化システム(第 4.8 節参照)は両者の方式の最もよい性質を利用している．しかしながら，そのシステムは，またそれぞれのシステムの欠点も多く持っている．MULTICS でそのようなシステムを実現したので，その可能性を示すことができた．実際，その番地付け方式の複雑さが，それによって得られる便利さによって正当化されるかどうかということはまだ解決されていない問題点である．

演習問題

5.1 第 5.2.1 節で論じた厳密な記憶の階層の概念に対して，いくつかの例外が存在する．たとえば，情報をテープからディスクに転送するときは主記憶を介してはじめて行なわれる．別の例として，あるCDCの機械では，情報をディスクから低速コアに転送するときは高速コアを介して行なわれる．このような矛盾に対する理由を述べてみよ．

5.2 一つの処理装置が記憶の管理をし，もう一つの処理装置がユーザの計算だけを行なうような計算機システムについて概略を述べよ．

5.3 仮想記憶の大抵のモデルは，要求ページ化を扱っている．一方，多くのシステムでは，ジョブを開始するときにある種の前ページ化を行なってい

る．なぜこのような矛盾が存在するのか．

5.4 プログラムの局所性の形式的概念について，概略を述べよ．多数の処理装置によって共有しているデータ構造に対してどのように局所性を定義するか．プログラム・モジュールとデータ・モジュールとは同様の局所性を持つか．その解答に対する正当性を述べよ．

5.5 プログラムの局所性を悪くするようなプログラミング言語のいくつかの特徴を列挙せよ．書いたプログラムの局所性を改善するために従うべきプログラミングの規則の集合について，形式化しないで述べよ［Weinberg, 1971］．

5.6 プロセスが最高優先度先処理によってスケジュールされていると仮定する．処理装置をこれから受け取ろうとしているジョブを，優先するようなページ置き換えアルゴリズムについて概略を述べよ．最低の優先度のジョブのページをいつも置き換えているのでは，なぜ適当ではないのか．

5.7 FIFO ページ置き換えアルゴリズムは，実現するのに最も簡単で最も安くできるものの一つである．しかし，FIFO ページ置き換え則は，特にコアのすべてのページに全域的に適用するとスラッシングをひき起す．FIFO の原則をほんの少し変更しながら，スラッシングの問題を避けられるような簡単な実現上の変更を提案せよ．そのアルゴリズムの性能をLRUやOPTと比較して評価してみよ．

5.8 この章で述べたいくつかのページ置き換え則を考えよ．各アルゴリズムにおいて，最もよく動くプログラムの性質について述べよ．

5.9 コンパイラが，コンパイルしたコードをページの中に配置する方式は，ページ・フォールトの振舞に対して重大な影響を与える可能性がある．(たとえば McKellar および Coffman［1969］)．“頭のよい”コンパイラならば，ソース・コードを調べて参照パターンを予測することができるであろう．コンパイラがページづけをするときに従うべき良い経験的な規則を列挙してみよ．どうして，それらの規則が有効に働くのか，その検証をせ

よ．

5.10　時間に従って，ワーキングセットが非常に大幅に変化するプログラムとは，大よそどんなものか．どんな置き換えアルゴリズムが，そのようなプログラムに対して，特に悪い性能をもたらすか．どんなアルゴリズムが，特に良い性能をもたらすか．

5.11　オペレーティング・システムが，第5.3.1.1節に述べたような，相棒システムを用いて自由空間を管理しているものと仮定しよう．システム内のジョブが，偶然（または故意）に同じ領域を二度返したならば，システムはいつかは壊れてしまう．相棒システムを設計し直して，完全に誤りに対して保護されるようにせよ．（計算機には，記憶保護機構はないものと仮定せよ．）設計した誤り保護つきの相棒システムと先適合アルゴリズムとを比較せよ．

5.12　次のようなページ置き換えアルゴリズム（ここでは，少し簡単化している）は，ATLAS 計算機［Kilburn ほか，1962］に対して提唱されていた．

すべてのページ（コアに存在，非存在を問わず）に対して，システムはページが最後に参照してからの時間 t とそのページに対する前の2つの参照の間の時間間隔 T とを記録している．これらの時間は，すべて実行された命令の数を用いて測定する．ページの置き換えが必要なときには，システムは次のようなページを選択する：

1.　$t \geq T+1$

または，**1.** を満すものが存在しない場合，

2.　$T-t$ が最大で，かつ $t \neq 0$.

このアルゴリズムを用いる理由は何か．これは，プログラミングの様式に対して，どの位，敏感であるか．実際は，このアルゴリズムが LRU ほどうまく働かない理由は何であろうか．

5.13　仮想記憶システムにおいては，コア記憶は有効に割り当てなければならない．すなわち，プログラムが実際に必要としているよりも多くのページ・

フレームを与えることは許されない．スラッシングとは正反対のこのような状態をどのようにして検出するか．プログラムに自分専用の領域が与えられていて，ページの置き換えは，自分専用の領域で局所的に行なうものと仮定する．このような場合，各プログラムがそのワーキング・セット以上のページを受け取らないようにするためには，どんな規準が必要か．

5.14 ページ化の環境において，主記憶の使用に対して料金を課す方式について概略を述べよ．ページ・フォールトの振舞や大きなワーキング・セットの規模のような内部的な問題に対して，ユーザは料金を支払うべきであろうか．

第 II 部

O. S. の技法

本書の第I部で扱った項目は，概念的な性質を持ったものであった．オペレーティング・システムは，システムの主要な資源を使用しながら，共同して働くプロセスの集合と見做すことができる．提示した概念は，たとえばプロセスとか記憶管理のように基本的でかつよく理解されているものである．

さらに，オペレーティング・システムは歴史的な理由から存在している側面やまだよく理解されていない多くの新しい側面などいくつかの特徴を持っている．ファイル・システムは前者の例であり，設計の問題は後者の例であろう．純粋な立場からすれば，これらの問題は，すべて避けて通ることもできよう．これらの話題を議論するのは，その材料に多くの注意を向けさせることに価値がある．恐らく少数の新しい概念が発見されて，これらの問題が統一化され，二，三の基本的なアイデアに昇華させられるのであろう．保護の話題は，ちょうどそのような位置にあると思われる．本書の残りの5章は，それだけで終ってしまい，うまく構造化されてないように見えるかもしれない．しかしながら，それを適当に記述できないということからその重要性を判断しようと思ってはならない．現在，発展中のアイデアを説明するのは非常に難しいのである．一方，それらの問題はオペレーティング・システムと言われるときの中心の問題と直接に関係することであるので，それらを無視することは公平ではない．

第 6 章

入出力とファイル

6.1 はじめに

ファイルは，ユーザに透明な方法で名前を管理し，データを配置するもう一つの技法を提供している．**ファイル**（file）とは，対象物の集合についての情報を物理的に表現したものである．ファイルは，通常，**レコード**(record) と呼ぶもっと小さい単位から成り立つが，それはそれぞれある特定の対象物の情報を持っている．対象物の属性は，レコードの**フィルード**（field）の値としてコード化してある．たとえば人事ファイルを考えてみよう．各被雇用者に対し，その人を記述するレコードが１つづつある．各レコードのフィールドには，名前，住所，配偶者，子供，部門，部長，職歴，給与などが記述されている．

ファイル・システム（file system）は，ファイルを実現するオペレーティング・システムの構成部分である．ファイル・システムが理想的には備えているべき性質が，７つある：

1. 二次記憶を有効に割り当てる能力.
2. アクセスの柔軟性および多様性.
3. できるだけユーザには見えないような機構.
4. できる限り機械や装置から独立であること.
5. 共通ファイルを共有する能力.
6. ファイルに格納してある情報の安全性および保全性.
7. ファイル操作コマンドの効率的な実現.

ファイル・システムは，基本的には異った２つの機能を備えている．第一に，

記憶装置においてデータを能率の良い方法で処理することに責任を持っている．第二に，ファイル・システムは，ユーザに対し自分自身の名前の空間を作り出し，柔軟な方法でそこにデータを貯え，そこから検索してくる能力を与えている．残念なことに，ファイル・システムの物理的および論理的な性質がときどき混乱していることがある．このような状況が，一貫したファイル・システムの概念の発展を妨たげてきた．本章では，二次記憶装置においてデータを構成するときの問題を集め，それを理解し易いグループに分けながら論じてみよう．

ファイルは，一般にディスクやテープ装置のような周辺装置に格納されている．ファイル・システムが解決すべき第一の問題は，これらの装置で記憶を効率よく割り当てることである．普通，周辺装置の物理的空間は，固定長または可変長の**ブロック**（block）単位に割り当てる．システムがページ化してあれば，断片化を避けるためにブロックの大きさとページの大きさと何らかの関係があることは重要なことである．ブロックの大きさとファイル内のレコードの大きさとは，関係があっても無くてもよい．

各ファイルは，本質的にはブロックの集合である．ブロックは，結合リストを形成して一緒に連結しておくことができる．すなわち，各ブロックはリスト内の次のブロックを指し示すポインタの領域を持っている．別の方法として，**記憶マップ**（storage map）という表を管理することができる．各ブロックに対し，ファイル内のブロックの位置を決めるエントリがそのマップの中に存在する．この第二の方法は，ブロックを処理するのに必ずしもアクセスする必要がないので，都合がよい（ディスクへのアクセスは，コアへのアクセスの1000倍は遅く，ディスク内でポインタを追いかけることはきわめて費用がかかる）．

システムを開始したとき，使用可能なすべての記憶は**空きプール**（free pool）すなわちまだどのファイルにも割り当てていないブロックの集合の中へ入れておく．ファイルを作成したり拡げたりすると，ブロックを空きプールから除去し，いろいろのファイルに割り当てる．ファイルを縮少し除去すると，そのブロックは空きプールへ戻す．コアの割り当て方式は，第5章で議論してきた．同様の方法がここでも用いられる．

ファイル・システムが解こうとしている第二の重要な問題は，名前づけの問題である．ユーザは自分自身の名前の空間を作りたいと思う．ユーザがファイ

ルを共有しているときは，そのファイルを自分の個人用の記号名を用いて参照したがる．そのような環境においては，単一の名前が異ったファイルを指すことがあり，異った名前が単一のファイルを指すことがある．名前が用いられている文脈の中で，名前づけの競合を解決しなければならない．

ファイルとは，本質的には名前をつけた記憶セグメント［Organick, 1972］である．ある文脈で用いられるファイル名は，番地からファイル内の対象物への写像を行なう関数を定義している．ファイルの方が，名前のつけられたセグメントよりもずっと柔軟にユーザの名前の空間を構成することができる．レコードは名前の空間の単位であり，それらへの参照は異った法則に従って実施することができる．すなわち，名前の空間は名前の間の関係によって作られる構造を持っている．たとえばユーザは，ファイル内のレコードを逐次的順序，木構造，あるいはユーザの応用に適した他の方式を持ったものと見ることができる．ファイル・システムは，また，第4章および第5章で述べたような名前－番地づけの関係を，単に主記憶－ページ化ドラムの対よりも，もっと多くの装置に拡張したものと考える．残念ながら，ファイル・システムは，歴史的にはよく知られている記憶管理の理論ができる以前からあらわれているので，第4章で示したようなセグメント化とかアドレス変換という術語を用いて記述してあることは稀である．

ファイル・システムでは**ディレクトリ**（directory）を構成することにより，文脈の中で名前を確立する．ユーザは記号名を用いて，特定のファイルを参照する．与えられた文脈の中の各記号に対して，記号名をファイルが存在する実際の場所に変換するためのエントリがディレクトリの中に存在する．この変換は，一つ以上のレベルのディレクトリをたどることによって行なわれるかもしれない．さらに，ディレクトリはそのファイルの使用に関する情報を持っている．たとえば，一つのファイルに対するディレクトリのエントリは次のような情報を含んでいるだろう：

1. そのファイルにユーザが与えた記号名．
2. そのファイルの実際の場所．
3. そのファイルの全てのユーザに対して同一であるような一意的な識別名．

4. ユーザに許してあるアクセスの種類.
5. 最後に，そのファイルにアクセスした時間.
6. どれだけ頻繁に，そのファイルにアクセスしたか.

例として，多くのユーザが共有している人事ファイルを考えてみよう．ユーザは，そのファイル内の特別なレコードをアクセスしたいものと仮定しよう．そのユーザは，そのレコードを指定するためにファイル内の位置（たとえば，第七番目のレコード）を用いるか，またはレコードを一意的に識別するためのレコード内のフィールドのある値［たとえば，氏名と，社会保障番号（social security number)］を用いる．（位置で参照すると，プログラムはファイルの構造に大きく依存するので，何らかのファイルの共有をするのには不利になる.）

要求を処理するために，ファイル・システムは名前のついたファイルの場所を捜さなくてはならない．ユーザが与えるファイル名は，ファイルの一意的な識別名を得るためにユーザ・ディレクトリによって定義される文脈の中で用いる．テープのリールやディスクのパックのような情報を格納している物理的な媒体は**ボリューム**（volume）と呼ぶ．一意的な識別名を得ると，ファイル・システムは，この識別名に付与されている特定のボリューム（複数個のこともある）の場所を捜す．ファイルを格納しているボリュームが装塡されていなければ，ファイル・システムはそのボリュームをどこかあいている駆動装置に装塡するようにオペレータに指示する．これを行なうと，ファイル・システムはそのファイルにアクセスすることができる．必要なレコード（複数個のこともある)の位置は番地として明白に与えられることもある．また，フィールドの値に基づいた計算を行なって，間接的に与えられることもある．あるいはファイルの構成に基づいて探索をすることにより位置を決めることもある．次にファイル・システムは，入出力のハードウェアに対して，適当なコマンドを作り出しファイルを格納している周辺装置と主記憶との間で必要なレコードの転送を行なわなければならない．一つ以上のレコードを要求するときには，時にはバッファリングが必要になることがある．

前の議論からあきらかなように，たった一つの要求を処理するためにもファ

イル・システムは，分くの多様な操作を実行しなければならない．ファイル・システムが実行する操作は，ユーザが自分だけで用いている名前から一意的なファイルを識別するというような本質的に論理的なものがいくつかある．また情報のブロックを，主記憶と周辺装置の間で転送するようなハードウェアに非常に密接した操作もある．ファイル・システムのこのようないろいろの機能は，ソフトウェアの一連のレベルによって記述することができる［Madnick および Alsop, 1969］．ハードウェアに近い方から始めると，これらのレベルはファイル・システム内の5つのサブシステムに対応づけすることができる（**図6-1**参照）．各レベルは，その前のレベルの機能に基づいて，より柔軟な機能をいくつか実現している．最初のレベルは，主としてファイル・システムの物理的な側面を扱っている．最後のレベルは，主としてファイル・システムのユーザ向きの側面を扱っている．

6.1.1 入出力システム

物理的な装置の操作は，ソフトウェアの第一のレベルで統合する．このシステムを構成するプロセスは，ハードウェアに密着して動く．それらは，ボリュームを操作して，情報のブロックを主記憶と二次記憶の指定された番地の間で，読み込んだり書き出したりする．

6.1.2 基本ファイル・システム

このシステムは，ファイルの一意的な識別名を**ファイル・ディスクリプタ** (file descriptor) と呼ぶものに変換することを責任をもって行なう．ファイル・ディスクリプタは，ファイルの物理的な番地およびファイルの大きさを与える．ファイルが複数個のブロックから構成されているならば，ファイル・ディスクリプタはそのファイルを作り上げる異ったブロックへのポインタを持っている．基本ファイル・システムは，またファイルをさらに小さなサブ・ファイルに分割［または**サブセット化** (subsetting)］するコマンドも持っている．

6.1.3 論理ファイル・システム

論理ファイル・システムの機能は，ユーザによって与えられる記号名に対応

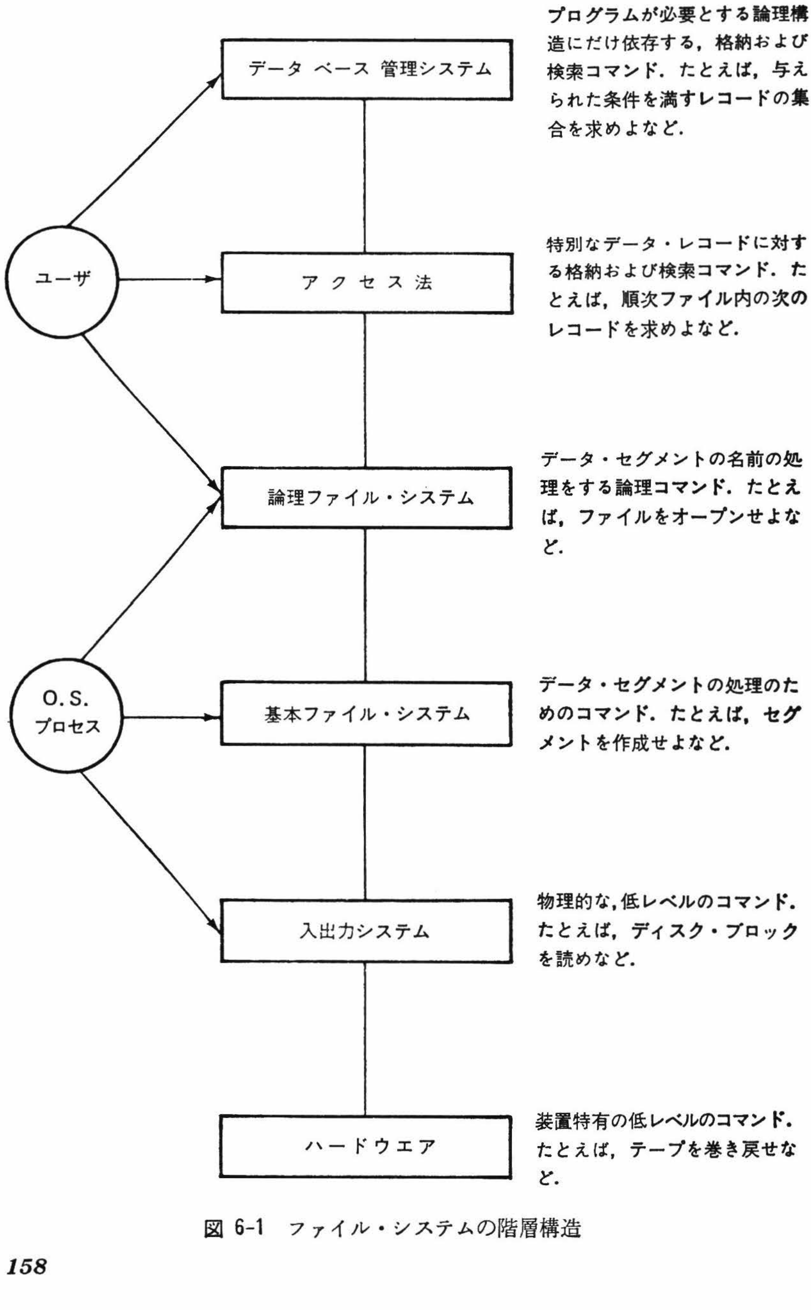


図 6-1　ファイル・システムの階層構造

している一意的な識別名を決定することである．記号名は，ファイルに付随している一意的な識別名を得るためにユーザのディレクトリの文脈の中で解釈するが，識別名は低レベルでの特定のディスクリプタに対応する．論理ファイル・システムの機能は，ファイルを格納している装置の物理的な性質とは無関係に行なう．このシステムでの処理を要求するユーザは装置と独立した形で行なう．

6.1.4 アクセス法

このシステムは，データを物理的に格納している方式とは全く異った方法で，データにアクセスする論理的な方法を実現している．ユーザは，時には，ファイルがボリューム上で物理的に配置している方式では直接表現できないような順序で，レコードにアクセスすることを望むことがある．たとえば，人事ファイルが社会保障番号の大きさに並べた順次的リストであると仮定しよう．**順アクセス法**（sequential access method）は，単にレコードを格納している順序でアクセスするだけである．しかし，ユーザの中には収入のようなフィールドの値に基づいてレコードにアクセスすることを望むものがいるだろう．これは，ユーザのアクセス法（たとえば，収入の大きさの順）をファイルの物理的構造に写像することにより可能となる．

6.1.5 データベース管理システム

実際にはファイル・システムの一部ではないけれども，データベース管理システムは，やはり自然に拡張して得られるものである．異ったプログラムは，処理しているファイルに対し異った論理構造を期待しているかもしれない．一方，含まれている情報が同一の場合，各構造に対して別々のファイルを保持しておくのは経済的ではない．データベース管理システムは柔軟なユーザ・プログラムとのインターフェイスを作り出し，単一の物理的ファイルに対していろいろの論理構造に従ってアクセスすることを可能にしている．この構成により，異ったデータの論理構造を必要とするプログラムの間で，同一の情報を共有することができる．

以下の節では，各レベルをさらに詳細に述べてみよう．ファイル・システムのいろいろのレベルによって与えられる機能は，低レベルのこまかいコマンド

から高レベルのユーザ向けのコマンドに致るまで変化に富んでいる．これらの機能をレベルに分割することにより，いろいろの処理をもっと管理し易いブロックに分けられるだけでなく，オペレーティング・システムおよびオペレーティング・システムのユーザ両方の要求を満すことがファイル・システムにとって可能となるのである．これら2つの異った機能は一つのファイル・システムの中で結合されているが，一般にシステム・プロセスを支えているファイル・システムの部分は非常に複雑なので，普通のユーザに対しては隠してある．

6.2　入出力システム

計算機の入出力ハードウェアは，通常，チャネル，制御装置および機器の階層構造になっている．入出力を起動する命令では，一般に機器および入出力操作を管理する内蔵プログラム［**チャネル・プログラム**（channel program）と呼ぶ］を格納している主記憶の番地を指定する．チャネルはチャネル・プログラムの命令を実行して，必要なデータの転送を実現する．操作の終了または異常状態の発生は，割り込みにより知らす．入出力割り込みが処理装置により受け取られると，入出力操作の状態を示す情報を格納する．誤動作が発生すると，構成要素の状態を調べる命令を用いることによりそれらの状態に関する追加の情報を得る．

入出力システムは，入出力ハードウェアの各要素に対応してそれを管理するソフトウェアのプロセスにより構成される．こうすることにより，第2章で述べたような標準的な相互通信の機構が，入出力システムの中で使用できることになる．それゆえ相互通信の路は，ハードウェアの結合の内部的なイメージになる．入出力システムの一つの機能は，処理装置と入出力ハードウェアとの間の相互作用を標準の相互通信形式に変換することである．こうすると，システムは，機器を協力して働くプロセスと見做すことができる．

この方式には，3つの大きな利点がある．第一に，ハードウェアにおける非同期的動作が，協力して働く独立のプロセスによって管理できるようにソフトウェアを構成することができることである．こうすると，機器間の速度の差はシステムにはもはや見えないので，残りのシステムの設計や実現が概念的にずっと容易になる．第二に，チャネルの誤動作のような特殊な状態が，ファイル

・システムの最も低いレベルで修正できることである．すなわちこれらの問題はそれらを処理するのに最も適したソフトウェアによって解決され，ソフトウェアの他のレベルに対して透明になる．第三に，すべての機器固有の処理を，ファイル・システムの最も内側のレベルに隔離することができることである．機器間の通信の微妙な点では，入出力操作を要求している他のプロセスに注意を払う必要はない．

入出力システムの主要な部分の一つの構成を次に示す［Atwood ほか，1972］．入出力システムを，機械と独立に議論することはほとんど不可能であるので，この構成は IBM のハードウェアの特徴にかなり影響されたものとなっている．

6.2.1 チャネル入出力システム

このサブシステムは，ハードウェアの各チャネルに一つづつ与えた**チャネル・マネージャ**（channel manager）と呼ぶプロセスから構成される．論理的には，各チャネル・マネージャはチャネル・プログラムを実行することを伝えるメッセージをチャネルに送ることにより入出力を開始する．チャネル・マネージャは，操作の終了または異常状態と指定したメッセージをチャネルから受け取りその入出力操作の終了を知る．チャネル・マネージャは，すべてのプロセスが用いるものと全く同一の通信の論理を用いて書くことができる．チャネル・マネージャと通信するプロセスは，あたかも制御装置に直接メッセージを送りそれから返事を受け取るかのように振る舞うことができる．したがって割り込みを整理し，チャネル状態の処理だけをし，他の情報を関連する制御装置マネージャに送り渡すことが，チャネル・マネージャの仕事である．

6.2.2 ディスク入出力システム

ディスク入出力システムは，各ディスク制御装置に一つのプロセスおよび各ディスク駆動装置に一つのプロセスを与えた構成になっている．チャネル・マネージャは（論理的）ディスクを操作する．各ディスクは，高レベルのコマンドを受け取るが，それをチャネル・マネージャの適当なコマンドに変換する．コマンドは，以下のようなものである：

1. システムの標準仕様に従ったブロックを用いて，ディスクの形式をきめる．
2. ディスク駆動装置上のボリュームを一意的に識別するようなボリュームのラベルを読んだり書いたりする．
3. 記憶からディスクに対して，情報のブロックを書く．
4. ディスクから記憶に対して，情報のブロックを読む．

同様の構成は，ドラムのような他の回転型装置に対しても使用することができる．

6.2.3　端末入出力システム

このサブシステムは，電話応答や切り換えサービスと同様な環境の中で，会話型の端末を支援しているものである．タイプライタ型の機器（処理装置のマスタ・コンソールも含める）は，すべてダイアルで接続できるような端末のように扱うことができる．したがって各端末は電話機のようにシステムにダイアルで接続し，その後システムから切り離すことのできるものとして振る舞う．システムに端末が接続されている間は，メッセージを送信したり受信したりすることができる．

チャネル・マネージャは，端末入出力システムを支援するために，制御用路と通信用路の2つのメッセージの路を構成する．制御用路は，主として端末がダイアルで呼び出すとき，または切り離すときに，システムに知らせるために用いる．通信用路は，何行ものデータを端末に送り出すときに用いる．

端末システムは，端末から（ダイアルの呼び出しの後に続いて）送ってきた最初の行を調べて，（もし必要ならば）どのサブシステムが端末と会話すべきかを決定する．そのようなサブシステムが，ちょうど一つ見つかれば，通信用路を，そのサブシステムに対して直接接続する．接続が完了した後では，端末からのメッセージはそのサブシステムに直接送信される．多くのサブシステムが関係している場合は，端末システムはメッセージが意図している各サブシステムをどのように区別し適当なサブシステムに転送するのかを決定する．

6.2.4　周辺機器入出力システム

周辺機器入出力システムは，自分自身のサブシステムによって管理するようなディスク，ドラムおよびタイプライタ型機器を除いたすべてのハードウェアの装置を管理する．周辺機器は，操作の特徴が非常に変化に富んでいるので，別々の扱いを必要とする．周辺機器入出力システムは，チャネル・マネージャを用いて次のような周辺機器を支援する：

1. 記憶から n 行プリントせよという形式のコマンドを受け付ける**プリンタ** (printer).
2. n 枚のカードを記憶に読み込めという形式のコマンドを受け付ける**カード・リーダ** (card reader).
3. n 枚のカードを記憶からパンチせよという形式のコマンドを受け付ける**カード穿孔機** (card punch).
4. 次のような形式のコマンドを受け付ける磁気テープ駆動装置.
   (a) 前進または，巻き戻しせよ.
   (b) テープから記憶に，n ブロック読み込め.
   (c) 記憶からテープへ，n ブロック書き出せ.

各構成要素は，それを管理するプロセスを持っている.

与えられた入出力システムの構成は，決して完結しているわけではない．入出力システムには，もっと詳細な事柄がたくさんあるが，その大部分については議論していない．読者にとっては，入出力システムが何を提供してくれるのかを理解することが重要である．これがどのようにして行なわれるのかを議論するためには，ある特定のハードウェアの特性について，ながながと詳細に述べて初めて可能となる．それゆえオペレーティング・システムの原理について述べている本で，それらの特性を深く議論することは適当ではないであろう.

6.3　基本ファイル・システム

入出力システムを介せば，システムの他の部分では，物理的な機器からデータを受けたり，そこへ送ったりすることができる．データを転送する前には，主記憶の番地と機器の物理的な番地とを指定する必要がある．物理的な番地を

指定するために，**基本ファイル・システム**（basic file system）は，ファイル・ディスクリプタを管理してそのファイルに属しているブロックの物理的な番地を得る．さらにオペレータと連絡することにより，ボリュームの装塡を監督し，またファイルの部分集合を定義したりファイルの記憶領域の拡大などをするときに用いる．

ファイル・ディスクリプタ（file descriptor）は，ファイルを一意的に識別するための少量のデータ構造である．それは，一般に，そのファイルに属している各ブロックの物理的な番地と，そのディスクリプタを使用し処理する方法についての制御情報とを含んでいる．

ファイル・ディスクリプタを作成したり，処理したりするために，基本ファイル・システムに対する別々のコマンドを用いる：

1. CREATE DESCRIPTOR (*Allocation*)

Allocation によって指定される領域に対して，新しいファイル・ディスクリプタを作成せよ．CREATE DESCRIPTOR コマンドを陽に定義するかわりに，全ボリュームのファイル・ディスクリプタを，そのボリュームを所有しているプロセスに自動的に与える．そのディスクリプタは，以下のコマンドを用いることにより，領域に応じたファイル・ディスクリプタの部分集合にわけることができる．

2. SUBSET (*File Descriptor*, *Subset*)

たとえば，先頭と終りの語で示した *Subset* で決まるファイルの部分集合に対応した新しいファイル・ディスクリプタを作成せよ．このコマンドを実行した後では，新しい *File Descriptor* をもとのファイルの部分集合を参照するために使用する．

3. ALLOCATE (*File Descriptor*, *Allocation*)

File Descriptor によって示されるものと，*Allocation* によって示される別の領域との結合空間に対応する新しいファイル・ディスクリプタを作成せよ．

4. DEALLOCATE (*File Descriptor*, *Allocation*)

File Descriptor によって指定されるファイルの一部分を，空き領域に返せ．

解放される部分は，*Allocation* によって，指定される．

基本ファイル・システムは，二次記憶のボリュームの管理も行なう．したがって装塡の指令を出すために，オペレータ・コンソールへの通信路を確立しなければならない．それを用いれば，オペレータにある特定の駆動装置にボリュームを装塡したり，どこか使用可能な駆動装置にそれを装塡することを指示できる．後者の場合，基本ファイル・システムは，どの駆動装置が選択されたか教えてくれることをオペレータに期待している．

基本ファイル・システムは，システム内のすべてのボリュームの現在の状態を書いてあるボリューム表を維持している．一つのボリュームに対するエントリには，次のような項目が書かれている：

1. ボリューム名と，一意的なボリュームの識別名．
2. そのボリュームが装塡されている場合は，その駆動装置の論理的名前．また，そのボリュームが装塡されていない場合は，フラッグ．
3. そのボリューム内で現在アクティブであるファイルの数．
4. そのボリューム内の配置を記述している表へのポインタ．
5. そのボリュームに関する保護情報（たとえば，所有者と正規のユーザ）．
6. 作成日付．

ボリュームを管理するために，基本ファイル・システムは，以下のようなコマンドを実現する．

1. REQUEST (*Volume Name*)

これを呼び出したプロセスは，システムがボリューム表に新しいボリュームのエントリを作ることを要求する．基本ファイル・システムは，所有権，料金計算，保護情報をチェックして，そのボリュームに対するファイル・ディスクリプタを作成する．*Volume Name* は，装塡の際の情報に用いるボリュームの局所的な名前でよい．それは，ボリュームの一意的な識別名と必ずしも一致する必要はない．

2. RELEASE (*Volume Name*)

基本ファイル・システムは，*Volume Name* で指定されるボリュームを，アクティブな使用から解除し，そのボリュームの所有者に関する料金計算の情報を更新する．このコマンドを発するプロセスは，そのボリュームの所有権を示すことを要求される．

3. ATTACH (*Volume Name*)

基本ファイル・システムは，*Volume Name* で指定されるボリュームを一つの駆動装置に結合する．駆動装置の所有権，または駆動装置を要求する権利をプロセスは示さなければならない．そのボリュームの状態が変化したことを示すために，ボリューム表を更新する．そのボリュームが装塡されていなければ，オペレータとの連絡が必要となる．

4. DETACH (*Volume Name*)

基本ファイル・システムは，ボリューム表を参照して，*Volume Name* で指定されるボリュームを駆動装置から切り離す．それから，その駆動装置が使用可能であることを，オペレータに教える．

前に並べたコマンドにより，基本ファイル・システムは2つの重要なサービスを与えることが可能となる．第一に，ファイル・ディスクリプタによってあらわされるようなボリュームの領域（基本ファイル）を管理する．第二に，ボリュームを管理して，ボリュームの結合および解放やシステム内に新しいボリュームを導入することができる．

基本ファイル・システムを用いれば，オペレーティング・システムは高レベルのコマンドに答えて，ファイル・ディスクリプタの処理を行なうことができる．こうすれば，ファイル・システムが(たとえば) READ コマンドを受け付けたとき，適当なコマンドを生成して入出力システムに与えることができる．また，ときには高レベルのユーザは，ボリューム上の特定の空間を参照したり，そうでなければそのような領域に関する情報を処理したりする必要がでてくる．ユーザに対してファイル・ディスクリプタの処理を行なわせるかどうかは，実際，設計上の選択の問題になる．ある場合，ユーザは，この機能を必要とするだろう．またある場合には，それは必要ないであろう．たとえば，ユーザは，

詳細なファイル・ディスクリプタよりも，ニモニックの記号名称の方を使いたくなるかもしれない．その場合，次節に示すような論理ファイル・システムが必要となるであろう．

6.4 論理ファイル・システム

論理ファイル・システムは，ユーザがつけた局所的な名前から，システムがつけた 一意的識別名に，名前の 変換を行なう（すなわち，第4．1節で述べた

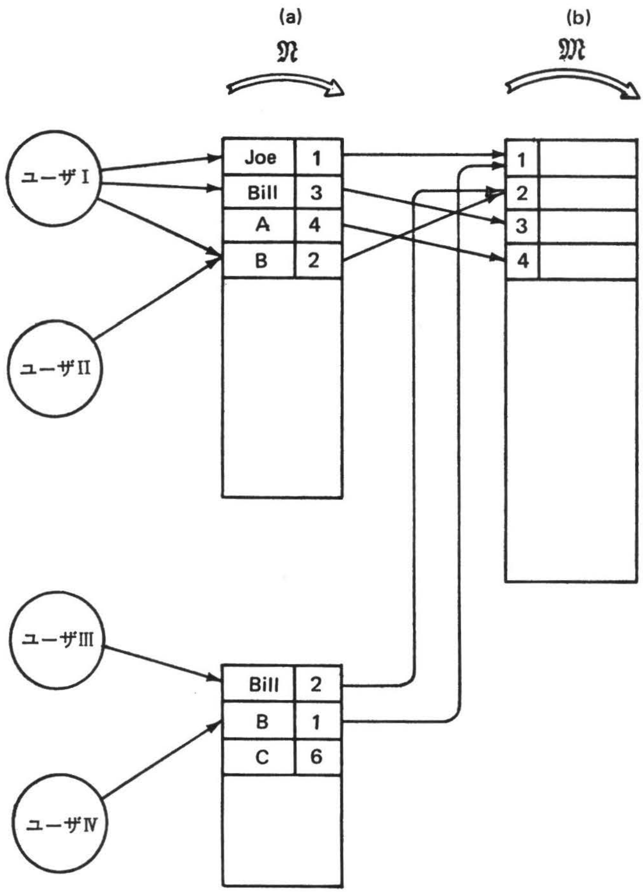


図 6-2　2レベルのディレクトリ構造．(a) スポンサ・ディレクトリ；(b) マスタ・ディレクトリ

関数）．さらに，基本ファイル・システムにコマンドを出して，読み書き操作のためのファイルを準備する．

論理ファイル・システムの各ユーザは，**スポンサ**（sponsor）と呼ぶエンティティ（entity）によって識別する．スポンサは，二次記憶に存在する恒久的な単位で，それは特定のユーザの料金あるいは料金用ファイルのリストに関する情報を含んでいる．ユーザが**ログ・オン**（log on）すると，システムはユーザのスポンサを識別するような**署名手続き**（sign-on procedure）から情報を得る．ユーザが**ログ・オフ**（log off）するときは，新しく生成したファイルのような，そのユーザの環境に関する恒久的な情報を，そのスポンサに書き写す．

スポンサは，ファイルに対してユーザがつけたニモニック名称を解釈するための文脈を与えるために使用できる．たとえば**図6-2**に示すように，各スポンサに対して別々のディレクトリを与えることができる．ディレクトリは，スポンサの1つのファイルを参照するためにユーザがつけた名前に対して，各々1つのエントリを持っている．ファイルに関する情報は，2つのステップにより得る．最初に，局所的な名前を用いて，スポンサのディレクトリから適当なエントリを見つけ出す．スポンサのディレクトリは，システム内のすべてのファイルのマスター・ディレクトリ内の一意的なエントリを指しているファイルの識別名を規定する．このようにして，各ユーザは他人と共有するようなファイルに対しても，自由に局所的なニモニック名称を用いることができる．対応する一意的な識別名を捜すことは，スポンサだけに関連することであり，したがって，かなり効率的に行なわれる．ファイルを共有すれば，データの不必要な複写を避けられるが，これは柔軟性のあるオペレーティング・システムの環境を作り出すために重要なことである．本章では，簡単に共有できる機構を強調している．第7章では，安全性の方針に従って共有を制限するような方法について，議論しよう．

ディレクトリに対するこのような2レベルの方式では，各ファイルに対して，一意的なエントリをマスター・ディレクトリの中に組み込んでいる．このエントリは，ファイルを識別するためのシステム全域にわたる一意的な名前，所有者の識別名，作成日附，アクティブなユーザの数またはその結合リスト，ファイルに属する領域に対するファイル・ディスクリプタなどを含んでいる．第6.

7節では，局所的およびマスタ・ディレクトリの考え方を用いたファイル・システムの例について，概略が述べてある.

論理ファイル・システムは，ファイルの作成，読み込み，書き出しなどをユーザができるようなコマンドを実現している．ユーザは，論理ファイル・システムに対して，ファイルに対して処理を施すというユーザの意図を知らせなければならない．この通知は，“open” コマンドによって行なう．open コマンドの結果，ファイル・システムは，ディレクトリ内のファイルを捜し出し read や write コマンドを受け付ける準備をする．読んだり，書いたりするデータの単位は，通常ファイル内のブロックの数で示す．ブロックは，必ずしもレコードに対応していなくてもよい．むしろ，それは二次記憶の割り当てのために用いる物理的なブロックと考える．論理ファイル・システムで実現するコマンドの例を，以下に示す：

1. CREATE (*File Name, Size, Volume Name* [optional])

File Name で示される新しいファイルを作ることを要求する．*Volume Name* が指定されている場合は，ファイルはその特定のボリュームに作らなければならない．保護や料金計算のための情報も，指定しなければならない.

2. DELETE (*File Name*)

File Name で示されるファイルを，削除することを要求する.

3. OPEN (*File Name*)

File Name で示されるファイルを開く.

4. CLOSE (*File Name*)

File Name で示されるファイルを閉じ，ユーザから切り離す.

5. READ (*File Name, Nfile, Mprocess, Count*)

File Name で示されるファイル内の *Nfile* ブロック目から，*Count* 個のブロックを読み込むことを要求する．これらのブロックは，*Mprocess* 番地から始まる記憶の連続領域に置かれる.

6. WRITE (*File Name, Nfile, Mprocess, Count*)

File Name で示されるファイル上に，*Count* 個のブロックを書き出すことを要求する．書き出しは，ファイルの *Nfile* ブロック目から開始する．主記憶内のデータは，*Mprocess* 番地から始まる連続領域内に置いてある．

各コマンドは，多くの結果をもたらす．たとえばCREATE コマンドは，以下に示す結果のうちの一つを引きおこす：

1. *File Name* というファイルが，指定にしたがって作成される．ユーザは，以後このファイルを参照できる．
2. *File Name* という名前は，スポンサがすでに持っている名前と衝突する．システムは，同一の名前を持つ別のファイルを作ることはできない．
3. *Volume Name* が見つからないか，装塡されていないか，または一杯になっている．ユーザは，自分のファイルのために別のボリュームを指定しなければならない．

ユーザのプロセスでは，これらの異った結果を扱えるような論理構造を持っていなければならない．そうでないと，これらの条件は予期しない誤りとして扱わなければならなくなる．

6.5 アクセス法

前節では，ファイルは，情報のブロックの集合と見做して議論し，それが持っている情報の性質や構造については関心を払わなかった．ユーザの各グループは，自分自身の特別な方式で構成した情報が必要となる．ユーザは，レコードのデータ・フィールドのある部分によって識別するような特定のレコードを，ファイル内で頻繁にアクセスする必要がある．このような機能として，システムは**アクセス法**（access method）を提供してくれる．

ユーザが興味を持っている特定のファイルにおける特別のレコードまたはレコードの集合を識別する方法は，いくつか存在する．最も簡単な方法は，ファイル内のレコードの位置を用いる方法である．論理ファイルが，レコードの順序を暗黙のうちに持っていれば，i 番目のレコードを番地づけすることは意味を持つ．（レコードの論理的な順序は，ファイルが格納されている物理的な装

置の性質とは無関係に決めることができる.）同様にして，ポインタが特定のレコードを示しているならば，“次”，“前”，“現在”ということは，そのレコードの意味のある属性になる．レコードの大きさは重要ではないので，これらの属性は単純な関係になる．レコードを参照するもっと柔軟な方法は，データのあるフィールドを**キー**（key）として用いることである．キーの値により，ファイル内でレコードを一意的に識別する．たとえば人事ファイルにおいて，社会保障番号はキーとして用いることができる．ある値を持つレコードが全然ないか，一つまたはそれ以上あらわれるようなデータのフィールドを**部分キー**（partial key）という．たとえば人事ファイルにおいて，給与フィールドは，300 万円の給与をとる社員は何人もいるので，部分キーになる．したがってデータの任意のフィールドは部分キーとして使用できる．レコード内のある特定のデータ・フィールドを，そのフィールドの値に関してアクセスするようにキーまたは部分キーとしてしばしば指定する．たとえば社員レコードを社会保障番号にしたがって検索するならば，それはキーとして宣言しなければならない．

質問語（query）は，ユーザがデータの検索の要求を指定できるようにするための言語要素である．質問語をデータの集合を構成するために用いたキーを使って書くならば，システムはそれらのキーを用いて，キーが用いられない場合よりもずっと速く特定のレコードを検索することができる．データの指定の仕方を質問語で実現するには，多くの方法がある．

第一の方法は，システムがレコードを順次的に走査して，キーの正しい値を持っているレコードを全て見つけ出すという方法である．この操作は，ファイル全体を走査しなければならないので，ディスクへのアクセスが大量になり費用がかかりすぎる．一方，プログラムを作るのには，非常に簡単である．また小さいファイルに対しても適している．

第二の方法は，キーの各値に対して，キーが適合するようなすべてのレコードに対するポインタを持つ要素をシステムが用意することである．これらの要素の集合は，それ自身ファイルであるが，**逆ファイル**（inverted file），または，**索引表**（index table）と呼ぶ．たとえば人事ファイルが，“部”を部分キーとして使用するものと仮定しよう．各部に対して，“部”逆ファイル内のポインタの集合は，その部に属するすべての社員を指し示している．この場合，部逆

ファイルは，各部に対応してグループに分けているポインタの集合で構成されている．ある特殊な質問語が，特に逆ファイルが作られているようなキーを用いたものであれば，順アクセス法よりもずっと速く検索できることになる．しかし逆ファイルそれ自身，非常に大量の記憶を必要とするので，適当な部について検索することが必要である．さらに逆ファイルは，その保守が面倒であり，特にキー・フィールドに於ける更新操作が多くなると，それが顕著になる．

第三の方法は，特定の部分キーに関して同一の値を持っているようなすべてのレコードのチェイン〔または**多重リスト**（multilist)〕を，システムが作ることである．たとえば，人事ファイルにおけるX部の最初の社員は，X部の次の社員へのポインタを持ち，これを最後のレコードまで続ける．チェインは任意に結合することができるが，ユーザには透明になっている．これらのポインタは，システムが作り，質問語を高速に処理するために用いる．たとえばユーザがX部の社員レコードをすべて要求したときは，システムは辞書を介するか，または走査することによって第一のレコードを見つけ出し，それ以後，残りのものを検索するためにポインタを用いる．この操作は，やはりある種の質問語に対しては速くなる．一方，これまでに述べたポインタはすべて空間を占有し，またレコード内でのキーのデータ・フィールドを変更したときには，注意深く扱い更新する必要がある．

実際に存在する環境の中で，前述した3つの方式のうち，どれを選択すべきかを決めるのは非常に難しいであろう．システムの性能は，処理する質問語の形式や頻度に大きく左右される．解析的手法，シミュレーション，モニタリングなどを用いて，各々の解法を評価しなければならない．しかしながら非常にはっきりと定義されている応用を除いては，実現のために最初に選んだ特徴は予測より大してよくならないであろう．

オペレーティグ・システムは，通常，標準化したいくつかのアクセス法を提供する．それらを用いれば，ユーザはファイル・システム内の走査や索引の機構を呼び出すような簡単なコマンドを用いて，ファイル内のレコードを識別し検索することができる．実際の実現上の機構は，ユーザには透明になっている．

オペレーティング・システムが提供するアクセス法は，2つの基本的な性質を持つ．それは**基本**（basic）または**待機**（queued）と，**順次**（sequential）ま

たは**直接**（direct）である．**基本アクセス法**は，IBM の BSAM[1] のように，ファイルから検索したレコードに対するバッファリングを自分で行なわなければならない方法である．この点に関しては，ユーザは自分の応用に適合するように自分自身のバッファリングの機構を工夫することは自由である．**待機アクセス法**では，システムはIBMのQSAM[2]のように，標準的なバッファリングの機構を備えている．ユーザは自分のデータをバッファリングする負担から解放されるが，システムが与える機構の約束事に甘んじなければならない．**順アクセス法**は，論理ファイルのレコードを順次的に構成する．ユーザはIBMのBSAMやQSAMのように，ファイル内のレコードを順番にアクセスして，データを検索したり格納したりする．**直接アクセス法**は，IBMのBDAM[3]のように，ファイル内のレコードをランダムにアクセスできる．ユーザは，レコードのファイル内での位置を考えることなく，望んでいるレコードを直接参照する．システムは，レコードの位置を決め，それにアクセスするために必要な機構を備えている．

直接アクセス法は，順アクセスの可能性を排除しているわけではない．通常，順アクセスも許しているが，順アクセス法を用いた場合のように効率的には働かないであろう．ある場合には順アクセス法を拡張して，たとえば，IBMのISAM[4]のように，索引を用いてファイルに対しある種の直接アクセスを行なうことができる．

アクセス法の属性（すなわち，基本，待機，順，直接）は，その詳細な実現法やファイルが存在している物理的な装置とは，概念的には無関係である．それらは，アクセス法がユーザに提供している機能の論理的な属性である．もちろん，直接アクセス法はディスクやドラムのような乱アクセス装置上のファイルに対して実現するのが，実際の現状である．順アクセス法は，一般に，レコードを記憶装置の中に順次的に割り当ててゆく．しかし，そうしなければならないというわけではない．とにかく，アクセス法の内部機構は，そのユーザにとってはほとんど関心がないはずである．残念ながらアクセス法は，個々の実

訳注．1) Basic Sequential Access Method
訳注．2) Queued Sequential Access Method
訳注．3) Basic Direct Access Method
訳注．4) Indexed Sequential Access Method

現法についての特別な事柄をあまりに強調し過ぎて提示されることがしばしば見られる．

実在するシステムの中で，多くのアクセス法がドキュメント化され，実現されてきた．大部分の応用プログラムは，オペレーティング・システムとこのレベルで接している．アクセス法は，伝統的には事務の応用に対して重要であると考えられてきた．

同一のオペレーティング・システムの中でも，多くの機能が重復している．

最近のいくつかのアクセス法では，いままで提供されてきたアクセス法の機能を再構成し，統合しようと努めている（たとえば，IBM の VSAM†)）．将来のハードウェアでは，実際，アクセス法のいくつかの機能を直接実現することになるかもしれない．

6.6 データベース管理システム

異ったユーザ・プログラムは，一般にそれらが処理するファイルの構造と形式について異った仮定をするものである．このために最近までは，ファイルの情報の内容が同一であっても，多数のファイルの存在することが必要であった．さらに対応するファイルに対してその形式を変更する度毎に，ユーザのプログラムを修正しなければならなかった．ファイルからのプログラムの独立性をある程度達成するために，また多数の異ったプログラムが同一の情報の一つの写しを共有するために，**データベース管理システム** (data base management system) と呼ぶ機能が開発されてきた．

データベース管理システムは，ユーザ・プログラムの必要性に応じて，いくつかの論理構造をある情報の上に重ね合わせるような応用システムである．もう一度，人事ファイルを考えてみよう．異ったプログラムは，人事ファイルを異った方法で扱う．たとえば，給与更新プログラムは，経営報告プログラムとは異った形式でファイルにアクセスする．ファイルを設計してしばらくしてから，新しい情報の必要性が生じてくるかもしれない．ファイルを修正するかまたは別のファイルを作成しなければならない．ファイルが，非常に大きく変化すると，古いプログラムは実行できなくなる．データベース管理システムは，

訳注．Virtual Storage Access Method

プログラムが処理をするファイルの構造に関して，プログラムがしなければならないすべての仮定を，そのシステムの一部分に集中している．要求を変更したり，要求がいろいろある場合などを考えて，データベース管理システムは，応用プログラムとの間に柔軟なインターフェイスを持っている．各プログラムは，データベース・システムを使用することにより，それが期待する“論理”ファイルを処理することができる．

データベース・システムは，ユーザが要求するいろいろの論理構造を実現するために，逆ファイル，索引，キー，多重リストなどを使用する．典型的なものでは，データベース・システムは，かなり大きくまた複雑なものになる．あるものは，全く独立した質問言語やデータ処理機能（たとえば，IBM の GIS[1] システム）を提供してくれる．また，あるものは PL/I, COBOL, アセンブラなどの親言語からの呼び出しの集合として実現している（たとえば，GE/Honeywell の IDS[2] や IBM の IMS[3]）．さらに，多数のユーザが，データベースに同時にアクセスできるようにするために，遠隔処理機能を提供してくれるものもある．そのような環境を，時には**経営情報システム**（management information system）と呼ぶ．

6.7 簡単なファイル・システムの例

この節は，ディスクを持ったミニコンに対して適しているようなファイル・システムを考えてみよう．このファイル・システムは，いくつかの点で UNIX システム［Thompson および Ritchie, 1973］に似ている．できるだけ機械に独立な設計をするために，入出力システムは議論しない．本質的には，非常に簡単な基本ファイル・システムの環境と，論理ファイル・システムに基づいた簡単なアクセス法の結合を例として示している．

各ユーザは，単一のスポンサに対応し，それ自身のファイルのディレクトリを所有している．スポンサのディレクトリのエントリは，ファイルの物理的番地を格納している**マスタ・ディレクトリ**を指しているポインタである．ユーザ

訳注．1) Generalized Information System
訳注．2) Integrated Data Store
訳注．3) Information Mnnagement System

・プログラムは，スポンサのディレクトリを用いてファイルにアクセスする．ディレクトリのエントリは，次の通りである（**図 6-3** 参照）：

1. ファイルに対して，ユーザ・プログラムで与えた記号名称．
2. マスタ・ディレクトリの i 番目のエントリを示す数 i．
3. 読み込みポインタ．

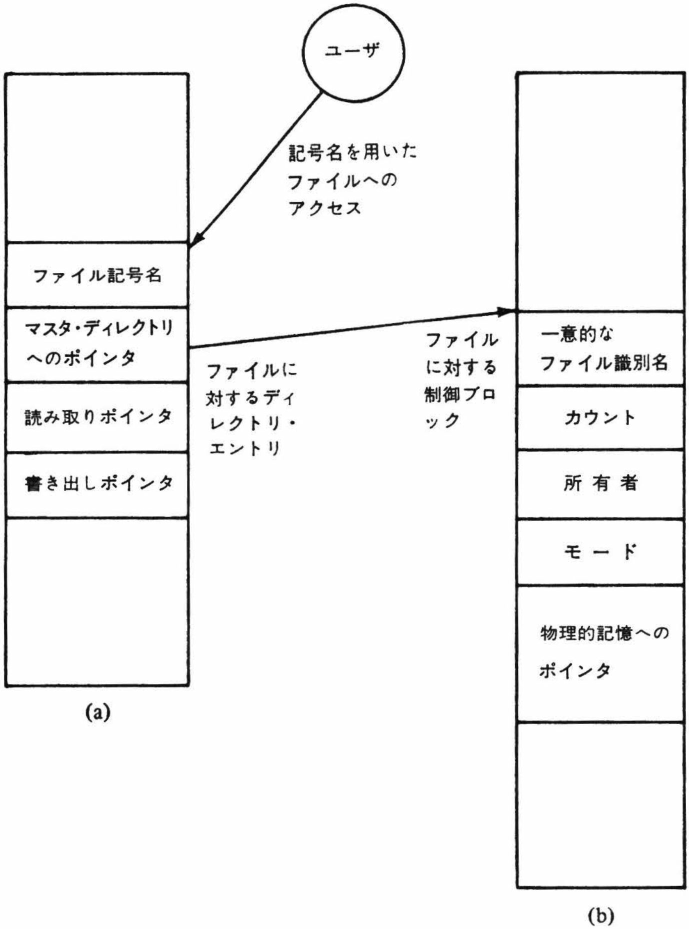


図 6-3　ファイル・システムの構造の例．(a) スポンサ・ディレクトリ；(b) マスタ・ディレクトリ．

4. 書き出しポインタ.

マスタ・ディレクトリは，ファイルに対する制御ブロックから成り立っている．各制御ブロックは（**図6-3**に示すように構成されており），次の情報を含んでいる：

1. 一意的なファイルの識別名.
2. 現在，ファイルをオープンしているユーザ・プログラムの数を示す**カウント**.
3. 所有者の識別名.
4. 所有者および他のユーザに許しているアクセスのモード.
5. ポインタの集団．各々は，情報を含んでいる記憶のブロックかまたは情報のブロックを参照しているポインタを含んでいる記憶のブロックを指している．各ポインタには，1ビット設けて，ポインタが間接的かどうかを示している.

記憶のブロックは64語の大きさであり，ポインタの集団には，高々，8個までのポインタがあるものと仮定しよう．これらの制限の下で，通常のファイルは64語から8×64語の範囲まで作成でき，また**長大ファイル**（long file）は，高々，8×64×64＝32K語までの大きさになれる.

ファイル・システムを介せば，ユーザはその読み／書きポインタを用いてファイル内の情報の語を示すことにより，それを参照することができる．ディレクトリの構造からすれば，ユーザはマスタ・ディレクトリを用いて共有するような共通ファイルの参照も，自分のディレクトリ内でニモニックな名前で行なうことができる．機能は，次のようなコマンドの形で提供する：

1. *K*:＝CREATE（*Joe*, *mode*）

このコマンドは，ファイルの作成の処理をする．引数 *Joe* は，ユーザ・プログラムでファイルを参照するために用いる記号名称である．システムは，マスタ・ディレクトリ内で空いている最初のブロックを見つけ出し，制御ブロックを作り出す．制御ブロック内のカウントは，1に設定する．一意的なファイルの識別名，所有者，モードなどを書き込む．一意的なファイルの識別名を，

変数Kに代入する．システムは，また，スポンサ・ディレクトリの中に，ファイル名 *Joe* のエントリを作成し（CREATE は OPEN も含む），読み込みと書き出しのポインタをファイルの先頭に設定する．

2. *Bill*: = OPEN (*K*, *Read*/*Write*)

マスタ・ファイルを走査して，変数Kに代入したファイルの識別名と一致するものを捜し出す．それが見つかれば，制御ブロックに与えてあるモードに対して *Read*/*Write* のチェックを行なう．もしも要求が許されているならば，スポンサのディレクトリにファイル名 *Bill* のエントリを作成し，カウントを1つふやす．このユーザ・プログラムは，このファイルを *Bill* として参照してもよろしい．ポインタは，適当に初期設定してある．たとえば，読み込みポインタはファイルの先頭に設定し，書き出しポインタは最後に設定してある．

3. CLOSE (*Joe*)

スポンサのディレクトリの *Joe* という名前のエントリを削除し，*Joe* に対応する制御ブロックのカウントを1減らす．もしもカウントが0であれば，その制御ブロックを削除する．最後の close コマンドは，delete コマンドとして働くことに注意しよう．独立した delete コマンドも，もし必要であれば，実現することは可能である．

4. *nread*: =READ (*Joe*, *buffer*, *read count*)

このコマンドの目的は，ユーザ・プログラムがオープンし，名前 *Joe* と一致したファイルから情報を読み取ることである．引数 *read count* は，読み込む最大語数を与え，また *buffer* は語を転送するバッファを指定する．読み込みは，現在の読み込みポインタから開始し，*read count* 語目に達するか，またはファイルの最後に達したときに終了する．n語（*read count* を越えない）ファイルから読み込むと，*nread* は値nをとり，読み込みポインタをnだけ増加する．

5. *nwrite*: =WRITE (*Joe*, *buffer*, *write count*)

これは，ファイルへ書き出しをするために用いる同様なコマンドである．*write count* は，ファイル *Joe* に対し *buffer* から書き出す語の最大数を示

す．実際に書き出した語の数は *nwrite* に代入し，書き出しポインタをその数だけ増加する．事前に seek コマンドで調整しておかない限り，書き出しポインタはファイルの最後を指していることになる．したがって，書き出しが始まると，ファイルが大きくなり重ね書きは生じない．

6. SEEK (*Joe*, *offset*, *flag*)

このコマンドは，読み込みや書き出しのポインタを，*offset* の値だけ調整するために用いる．引数 *flag* は，次のような意味を持った4つの価のうちの一つを仮定している：

flag＝0： 読み込みポインタを，ファイルの先頭から調整する．
flag＝1： 読み込みポインタを，現在の位置から調整する．
flag＝2： 書き出しポインタを，ファイルの最後から調整する．
flag＝3： 書き出しポインタを，現在の位置から調整する．

ファイルをアクセスし，*flag* の値によって読み込みまたは書き出しポインタの位置を決め，*offset* の値だけ調整する．*flag* を0または2に設定すれば，ファイルの先頭または最後に調整できるが，これはポインタの位置を追跡するよりも，しばしば容易であることが多い．

7. *ntell*:＝TELL (*Joe*, *Read*/*Write*)

ファイル *Joe* の読み込みまたは書き出しポインタの現在の位置を，変数 *ntell* に代入する．

6.8 おわりに

ファイル・システムの基本的性質と，その設計に対する重要な基準を概説してきた．ファイル・システムをハードウェアから始まるソフトウェアのレベルの系列と見做して議論してきた．

1. **入出力システム：** これは，ハードウェアの特殊性をマスクし，メッセージの転送やブロックの転送を用いた入力-出力機能を実行する．
2. **基本ファイル・システム：** これは，ボリューム上の領域を割り当てたり，

ボリュームの管理をする．

3. **論理ファイル・システム：**　これは記号名称とユーザ向きの操作を可能とする．

4. **アクセス法：**　これは形式化したファイルを実現し，いくつかの機構を用いてユーザが各々のレコードを識別し，それにアクセスできるようにする．

5. **データベース管理システム：**　これはファイルの同一の物理的な写しに基づいて，いくつかの論理データ構造を提供する．

最後に，ファイル・システムの簡単な例を用いて，いくつかの技法を説明した．

ファイル・システムは，多くのオペレーティング・システムに組み込まれてきたが，初期の頃のテープ向きのシステムから始まり，数年にわたり大量で色どり豊かで，一般には不正確な言い伝えとして伝えられてきた．この矛盾する言葉の実体にある光を投げかけ，基本的なアイデアを記述するのには，概念の確立を必要とする．たとえば，基本的な計算の単位であるプロセスは，オペレーティング・システム内の操作を記述するために用いている．ファイル・システムにおいては，その主要な機能はデータの管理を行なうことであるので，基本的な概念はデータの構造を作るという観点から論じなければならない．残念ながら，データの構造についての基本的な概念については，まだ一般的な見解の一致が見られない．計算することをあらわすプロセスという概念に双対的になる，データの構造をあらわす概念が必要である．データの管理は，その基本的な機能や操作を理解するという観点からみると，計算という操作に比べて遅れている．その結果，この章は前の章に比べて，特殊でその場限りのように見えるかもしれない．それにもかかわらず，ファイル・システムは計算機科学の他の発展している分野に比べて，概念的には優雅ではないけれども重要な課題である．

演　習　問　題

6.1　ファイル・システムは，仮想記憶の粗末な代用品となるか．何か，余分

な機能を提供しているか．仮想記憶により，ファイル・システムの実現が容易になるか．答の正当性を論ぜよ．

6.2 ディレクトリは，どの位大きいと考えるか．それらは，常にコアに常駐しているか．ディレクトリを格納する別の方法を追及せよ．ディレクトリ内のエントリにアクセスするために，どんな技法を使用するか．実在するシステムからとった例を示せ．

6.3 各々の共有ファイルに対して，現在，そのファイルをオープンしているすべてのユーザに関する情報を，システムが持っていた方がよい．与えられた時間に，ファイルをオープンしているユーザの数だけを持っている場合と，一意的名前かまたはバック・ポインタを用いて識別するユーザのリストを持っている場合の相対的な利点を論ぜよ．

6.4 第10章に述べる SUE システムを考えよう．入出力システムの設計に対し，IBM/360 のハードウェアが影響を与えた点を指摘できるか．入出力ハードウェアの構成をどのように変えると，簡単な入出力システムが可能となるか．

6.5 ファイル・ディスクリプタの実現法の概略を述べよ．どんなハードウェアの特徴が，これらのディスクリプタの処理にとって望ましいか．設計したディスクリプタと， Burroughs の機械に見られるセグメント．ディスクリプタとを比較せよ．

6.6 オペレーティング・システムの内部でボリュームを管理しているプロセスは，ディスクのボリュームを物理的に装塡したり取りはずしたりするオペレータと，交信する必要がある．オペレータと，ボリュームを管理するプロセスとの間の会話の形式を設計せよ．

6.7 ファイルが含んでいるデータの共有は，写しを取ることによって可能となる．データの写しを取ることと，単一の共有ファイルとの相対的な利点を論ぜよ．写しをとることを用いれば，ファイル・システムの設計を，どの位簡略化できるか．

6.8 仮想記憶システムでは，各プロセスは，それが用いる仮想番地空間と同一視することができる．この場合，プロセスがアクティブでなくても，その仮想番地空間に手をつけないようにしてプロセスを退避させることができる．そのような環境のもとで，第6.4節に述べたスポンサの概念は適用できるか．その答の正当性を論ぜよ．

6.9 ファイル・システムでは，なぜ独立の“open”コマンドが重要であるのか．（“open”コマンドは，“read”または“write”コマンドに含ませることができる．）

6.10 IBM 360および370のシリーズで提供しているアクセス法について概括せよ．アクセス法で提供している機能について，何か冗長性があれば，指摘せよ．異った応用に適用する単一のアクセス法を選ぶとすれば，どれを選ぶか．その答の正当性を論ぜよ．

6.11 ユーザは，陽にポインタを用いるか，または陰に条件づけやキーを用いるかして，ファイル内のレコードを処理できる．たとえば，あるレコードを示し，それから NEXT または LAST のようなコマンドを用いて，そのまわりのレコードを選択することができる．別の方法として，ユーザはそれが必要とするレコードすべてを一つの文で条件づけすることができる．たとえば，給与のフィールドが15000ドルのレコードを指定する．システムは，それらのレコードを得るためにユーザに対して透明になっているポインタを使うことができる．2つの方法の相対的な利点を論ぜよ．

6.12 レコードのフィールドが，未定義を意味する“空”値をとるようなファイル・システムを考えよう．計算において，変数または式の可能な値として，空値を扱えるプログラミング言語の機能を述べよ．実行時のチェックは，コンパイル時のチェックよりもずっと費用がかかることを思い出せ．

6.13 データベース管理システムにおいては，データ処理のコマンドは，IBM IMS システムの COBOL のように親言語から派生するか，または IBM GIS システムのように言語として集大成されている．この2つの方法に

ついて，相対的な利点を論ぜよ．

6.14 いくつかの先導的なメーカーの機械で使用可能なファイル・システムについて，比較のための概括の準備をせよ．

6.15 データベース管理システムとその機能についての概括の準備をせよ．

6.16 実在するデータベース管理システム，たとえば，IBM IMS/GIS/AAS, TDMS, TOTAL, MRI2000 についての事例研究をせよ．

6.17 データ通信の機能と，その費用に関する概括の準備をせよ．

6.18 米国国防省 ARPA の推進している計算機のネットワーク†) についての事例研究をせよ．

訳注．†) ARPAネットワークと称し，全米の大学および研究所40数カ所を結ぶ大規模な計算機ネットワークである．

第 7 章

保　　　　護

7.1 は じ め に

システムは，ユーザ（またはプロセス）が故意または偶然に，他のユーザ（またはプロセス）を傷つけることを回避できなければならない．ユーザは，友好的でかつ誤りを犯さないと仮定するのは，現実的ではない．ユーザは，自分または他人のプログラムやデータを破壊したり，すべてのユーザが受けている処理の低下をさせることにより，害を及ぼす可能性がある（究極的な低下は，システムの“破滅”を招く）．

上述の文章では，ユーザはある計算をしたい人という意味である．システム内には，単一のプロセス，プロセスの集団，記憶のブロックなどとして，ユーザがあらわれてくる．本章では，“ユーザ”という言葉は，むしろあいまいに使用し，システム内でシステムの機能を用いながら，ある人の計算を実行しているような実体を示すことにする．

保護（protection）とは，システムの要素をその環境から保護する機構を記述するための一般的な術語である．よい保護機構であれば，ユーザが他のユーザに干渉するようなことは，許してはならない．さらにユーザが，そのプログラムやデータを自分自身に対して守るために役に立つ道具も提供すべきである．

保護機構は，システムによってさまざまである．実際，時には単一のシステム内でも，異った部分では異った機構がある．たとえば，あるシステムにおいてはファイル・システムに対して一つの保護機構があり，主記憶に対しては別のものがある．

計算機システムにおいては，すべての情報は何らかの記憶装置に存在してい

る．したがって，情報はこれらの装置の番地空間を領域に分割して，この領域間の情報の流れを管理すれば保護が可能となる．番地空間の各領域は，その環境の残りの部分が情報を得たり変更したりするときに，許可していない要求から保護しなければならない．典型的な例として，ファイル・システムの保護を考えよう．各ファイルは，許可してよい読み込みまたは書き出しのコマンドを発するようなユーザから，保護されていなければならない．

領域間の情報の流れを制御するためには，2つの基本的な問題が存在する．第一に，番地づけ可能な空間の領域のまわりに，通信のための特別な“入口”を持った“壁”を打ち建てなければならない．領域にアクセスする唯一の方法は，入口を通して行なうことである．すなわち，壁は通行不可能である．第二に，領域を越えた通信は“監視”しなければならない．すなわち，入口には制御機構があり，許可したユーザにだけその領域にアクセスすることを許す．番地づけ可能な空間の領域のまわりの壁は，番地づけ機構（たとえば，ページ化とかセグメント化）と関係していなければならない．効率を上げるために，領域内の番地の参照は厳しいチェックを必要とはしない．領域の外側の項目の参照は，入口のような機構を介して行なわなければならない．

たとえば，ユーザはアクセスを許されているような番地の集合を持っているものとする．そのプログラムによって生成する番地を，解釈しチェックする．基礎および上限番地を持っているようなセグメントを用いたセグメント化番地空間では，このチェックは直接実行できる．各番地の参照は，基礎番地および基礎番地と上限とを加えた番地との間に入っているかどうか，チェックする．番地づけ機構によって，基礎-上限の制限を越えるようなすべての番地に対して壁が作れる．

セグメントも，陽に保護することができる．セグメント・テーブルを介してセグメントを参照するたびに，その参照を生成したプロセスに対する保護状態のチェックを行なう．同様にして，プロセスがファイルをアクセスしようとすると，その要求をファイル・システムに伝え，ファイルの保護状態に対してそのプロセスの権利をチェックする．ファイル・システムを介さずして，ファイルにアクセスする方法はないようにしなければならない（すなわち，入口である）．ファイルは，番地づけ可能な空間の領域．または単に保護を必要とするシ

ステム内の単位と見なすことができる.

保護を必要とするシステム内の項目を**対象物**（object）という［Lampson, 1971］. 壁を作り出すもう一つの方法は，対象物への結合のパスを秘密にしておく方法である. すなわち，対象物は保護されていないが，その場所はある権利を与えたプロセスにしか知らせていない. 他のプロセスは，その対象物を番地づけするために内部的名称を用いることができるが，結合のパスは，権利のあるプロセスにだけ公開してある. このような機構は，壁が侵入される可能性があるので，一般には適当な保護の機構を与えることはできない. ユーザは，偶然または故意に対象物に到達し，それを破壊することがまだ可能である. しかしながら，仮想記憶の環境においては，番地空間が大きいと，このような干渉はほとんど起り得ないことになる.

壁を作る第三の方法は，コード化することである. 情報は保護されていない領域に存在するが，それはコード化してある. 情報をデコードするためには，プロセスはコードのキーを知らなければならないが，それは権利のあるユーザにだけ与えてある. コードは，情報への入口となる.

番地づけ可能な空間の各領域は，入口として働く**保護モニタ**（protection monitor）によって保護されていなければならない. モニタは権利のあるいくつかのプロセスだけに，入口を通ってその領域にアクセスすることを許す. 同期を処理するモニタ（第2.6.2節参照）と同じものを拡張して，保護を処理することが可能である. 領域の外側からのアクセスが生成される（すなわち，あるユーザが入口を通ろうとする）ときは，いつもそれは適当な保護モニタを通らなければならない˙ このモニタは，ハードウェアでもソフトウェアでもよい. たとえば，IBM/360 の機械における物理的な保護キー†) と実行中のプロセスのプログラム状態語内のキーとは，記憶の2 Kバイトの領域の保護をするハードウェアの保護モニタを形成している. 一方，ファイル・システムは，二次記憶上の情報の安全を守るソフトウェアのモニタである. 伝統的には，機械は精巧な保護のハードウェアを備えてはいなかった. すなわち，柔軟性のある保護機構はソフトウェアによってのみ実現できるのであった. しかしながら，将来の機械は，ハードウェアやマイクロ・プログラミングによって，基本的な保護機

訳注，†）p. 194 の訳注参照（記憶キーと保護キー）

構がもっと多く提供されることになるであろう．

保護モニタは，アクセスの試みを制御するために，対象物への結合パスのどこかに論理的には位置している．モニタは，対象物に局所的なものでもよい．すなわち，保護を必要とする各対象物は，自分自身の保護モニタを所有している．一般に，対象物のある大きなクラスに対してアクセスの規則は同一のことが多いから，局所的な構成は同じ努力をたくさん繰り返すことになる．たとえば，保護をチェックするために，各々のファイルが自分自身の機構を持つことは，明らかに効率が悪い．一方，システム内の全てのアクセスの試みをチェックするような中央の保護モニタもあり得る．しかしながら，各々のアクセスは単一の保護モニタを通過するので，この方式ではシステム内に重大な隘路をしばしば生ずることになる．この問題は，モニタをハードウェアで実現することにより，完全とは言えないまでもある程度解決することができる．

一つの魅力的な解決法は，グループに分けてモニタをすることである．各々の保護モニタは，同じ様な対象物の大きなクラスに対する入口として働く．典型的な例は，やはりファイル・システムであるが，これは，ファイルと呼ぶ対象物のクラスに対するアクセスの権利に関する規則を監督している．

アクセスしようとしているプロセスの身元をチェックすると共に，保護モニタは通過を許可するデータを調べることもできる．これにより領域を破壊するような悪いデータのチェックをモニタがすることができるので，保護領域に対して余分な保護の手段が得られる．しかし，明らかにこのチェックをするためには，モニタ内でオーバーヘッドの増加という犠牲を払わなければならない．

最も困難な保護の問題は，相互に疑いあっている2つのプロセス間の交信である．各プロセスは，もう一方のプロセスが生成するアクセスを制限しようと努め，それが受け取るデータを詳細に調べる．2つのプロセスが，相手との交信の必要性を相互に理解することが重要である．第三の独立なプロセスを，保護モニタとして働くように選んでもよい．このモニタは，プロセス間の契約に似ている．2つのプロセスは，スーパーバイザについての意見は一致し，以後，“高度な権威者”に伝えなければ，何も変更することはできない．

保護機構の意図的な違反の扱いについては，興味のある哲学的な問題につき当る．大抵のシステムでは，許可していないアクセスは単に拒否する．保護モ

ニタは意図な違反を記録し，恐らく，頻繁に違反するものを報告することができる．違反を試みるようなプロセスに対し罰金を課し，“監督する”オーバーヘッドの支払いを助けることは，非常にもっともなことだと思われる．

7.2 領分と資格

番地づけ可能空間の領域の概念は，システムの保護状態を記述するのには，あまりにも一般的すぎて役に立たない．保護機構の設計を理解するためには，ある概念が必要となる．保護された環境においては，受動的な要素と能動的な要素とが存在する．保護を必要とする要素を**対象物**という．対象物にアクセスしようとする要素を**主体**（subject）という［Graham および Denning, 1972］．プロセスはシステム内の典型的な主体である．能動的な単位として，プロセスは保護されている対象物に対する処理を要求する．ファイルは典型的な対象物である．主体と対象物の役割は，時間と共に，時々変化する．たとえば，2つのプロセスは相互に防御し合ってもよいが，この場合，両者とも主体と対象物とを交互にくり返している．

異った主体が，異った対象物に関して同等な権利を持っていることが時々あるので，権利のクラスを独立の実体として設けると役に立つ．**領分**（domain）は，アクセスの権利の集合として定義する［Lampson, 1969, 1971］．領分は，主体が処理を実行できる保護の背景を与えている．主体やプロセスに保護に関する権利を付随させるかわりに，領分に権利を付随させる．この環境のもとでは，各プロセスは，いつもどこかの領分の背景の中で処理を行なう．他のプロセスは同一の領分の中で処理を行なうことにより，同等の保護の権利を共有することになる．プロセスはそれが処理を行なっている領分を変更することにより，その保護状態を変更する．保護状態を領分として分離したことにより，システム内で処理を行なっている主体とは独立にアクセスの権利を管理でき，保護機構を共有するための非常に柔軟な環境が実現できる．

あるシステムにおいては，各主体に対し一意的な領分を付随させると便利なことがある．このような構成においては，主体としてのプロセスはそれらのプロセス・ディスクリプタにコード化された保護の権利を持つことができる．そのような環境の下では，かなりの柔軟性が失われてしまう．プロセス内の一つ

の手続きがプロセスの他の部分とは異った保護状態を必要とするときは，その手続きは別のプロセスに移さなければならない．単一のプロセス内では，2つの領分が区別できないので，別のプロセスが必要となるのである．

対象物，主体および領分は，一意的な識別名によって識別する．この機構は，保護モニタがアクセスの権利の正当性をチェックできるために必要となる．長いビット・パターン（たとえば，64ビット）を用いれば，数年間にわたり重復せずにシステム全体に一意的な識別名を作り出すことができる．能率のためには，用いられている特定の背景のもとでのみ一意的であるような識別名を用いるのも便利であろう．たとえば，プロセス識別名は他のプロセスに対しては一意的でなければならないが，それらはファイル識別名に対して一意的である必要はない．

対象物に対する主体の特定のアクセスの権利を**資格**（capability）と呼ぶ［Dennis および Van Horn, 1966］．資格は他の実体と同様に管理することができる．たとえば，ある特定のファイルに対する読み込みだけというアクセスは，一つの資格である．同じファイルに対する読み／書きのアクセスは，別の資格である．“資格”という言葉は，アクセスの権利をコード化している特定のデータ構造に対しても使用されるので，時々，混乱をひきおこす．資格という抽象的な概念は，その使用があいまいなときは，いつでもソフトウェアでの実現法とは区別しなければならない．

保護の権利の取り消しは，議論の多い問題である．主体の持っている保護の権利は，取り消すことができないものと仮定すると，ときには，便利なことがある．この場合，保護の権利が不正なものになってしまうと，主体は許可されていないアクセスをするという誤りを犯すであろう．別の見方として，保護の権利を発行するプロセスが，それを自由に取消すことを許すというやり方がある．たとえば，資格を発行する資源管理者が，管理している資源に付随している資格のすべてまたは一部分を，無効なものにしてしまうことができるものとする．資格を失った主体は，資格が侵害されないようにする［Lampson, 1971］かまたはシステム内の“高度な当局”に不満を訴えることによって，保護される権利を要求できる．

資格を用いた例として，プロセスが，高々n回対象物にアクセスする権利を

与えられたと仮定しよう．そのような制限は料金計算のためには，重要であろう．制限したアクセスの権利を認める方法として，いろいろのものがたくさん考えられる：

1.　プロセスには，“パス”の形で資格を与える．保護モニタは管理者として働くが，カウンタを維持し初期値を０とする．プロセスが資格を使用するたびに，カウンタを１づつふやす．カウンタがnまで達したときに，保護モニタは資格を無効なものとする．

2.　プロセスにはn個の資格を与えるが，それは“切符”の綴りとして扱う．アクセスを行なうたびに保護モニタは“切符”を１枚取り消し，その資格を無効なものにしてゆく．

3.　資格を実現しているデータ構造にカウント・フィールドを設けて，その初期値をnとする．資格を使用するたびに，カウント・フィールドを１づつ減らす．保護モニタは資格のカウント・フィールドが０になったとき，その資格を無効なものとする．この形の資格は使用するごとにパンチするような“切符”に似ている．

4.　数nが格納されているもう一つの資格への間接的参照をするためのポインタとして，資格のデータ構造を使用する．アクセスを行なうたびにその資格を用いて，“共通”の資格を間接的に参照し，それを１つ減少する．共通の資格が０になると，すべての間接的な資格は無効なものとなる．この方式によれば，多くの主体が共通の資格によって表わしている資源の共通のプールを共有することができる．すなわち，各間接的な資格が小切手帳であるような，共通の“当座預金”を持っているものと考える．

資格のいろいろな使用の仕方を，概念的に述べてきた．このような概念に基づいた実現方法は，第7.4節にまわす．次節では，システム内の保護の状態について述べよう．

7.3　保護状態の記述

システムの保護状態は，単純なモデルを用いて記述できる．ある与えられた時間に，領分の集合 $X=\{x(1), \cdots, x(n)\}$ と対象物の集合 $Y=\{y(1), \cdots, y(m)\}$

とがあるとする．**保護状態**（protection status）を写像 F：$X \times Y \rightarrow A$ で記述する．ただし $A=\{A(1), \cdots, A(s)\}$ は，読み込みだけ，読み／書きまたは実行だけというようなアクセスの属性の集合である．領分は主体の操作の特殊性に左右されることを避けるために，主体とは独立なものと考える．任意の時点において，与えられた主体を領分と関連づける機構があるものと仮定する．

保護状態は，決して静的なものではない．領分や対象物は，きちんと定義した規則に従い，連続的に生成され，消滅され，変更される．さらに，主体はそれが働く領分を変更する．モデルはシステムの任意の瞬間の状態を与えているに過ぎない．

写像 F(x, y) が x と y だけによって決まるならば，システム内の領分の名前を行に持ち対象物の名前を列に持つような行列によって，保護状態を記述することもできる［Graham および Denning, 1972］．各領分と対象物に対して，行列は対象物に対する領分の権利を定義する．権利は，読み込みとか，修飾というようなアクセスの属性を用いて表現することができる．領分はシステムの対象物の非常に少ないものに対する権利しか持っていないことが多いので，一般にこの行列は稀薄である．行列モデルはアクセスの権利を決めるのに対象物や領分の名前だけしか許していないので，関数モデルほど自由度があるわけではない．しかし，行列モデルは行列のエントリとして手続き（すなわち，他の変数の関数）を許すことにすれば，一般化することができる．

たとえば，1000個以下のレコードを含んでいるファイルに対してのみアクセスの権利を持っている領分を考えよう．関数モデルでは，アクセスを認める前に，写像Fがファイルの大きさをチェックできるように各ファイルの保護フィールドに，ファイルの大きさと名前とを書いておく．行列モデルでは，“小さいファイルに対するアクセス”を示すような特別の属性を導入しなければならず，それにより行列のエントリを修飾する．そのかわりに，行列のエントリに手続きを与えておけば，アクセスの試みはその手続きによって正当性を検証する．その手続きは，本質的には資格（たとえば，“小さいファイルへのアクセス”）に付随しており，多くのアクセスの制限条件を有効に実現することができる．

行列を用いた保護の方式では，行列のエントリを管理なしに変更することを

許してはならない．なぜならば，それを許せば保護は何もなくなってしまうからである．もしも領分の権利を自由に変更する必要があるならば，それは保護していなければならない．この保護は領分を対象物として扱うことにより，実現できる．行列を拡張して，領分に対応する列を作る．そこで領分をアクセスでき，変更できることになる．

この構成を用いたとき，保護モニタは領分の権利が成長してゆくことに完全に責任をとる必要はない．たとえば，領分の権利の変更を管理するための次のような方式を考えてみよう [Lampson, 1971]．アクセス行列Aは領分を行とし，領分と対象物を列として構成する．さらに，**所有者**（owner）と**制御**（control）と呼ぶ特別の権利と，各アクセス権に対する**コピー・フラッグ**（copy flag）とを与える．領分 $x(i)$ が他の領分 $x(k)$ に対して制御属性を持っているときは，$x(i)$ が $x(k)$ を完全に制御することを意味する．$x(i)$ が，yに対して所有者属性を持っているときは，$x(i)$ が対象物yを所有することを意味する．コピー・フラッグは，アクセス権を他の領分にコピーしてもよいことを示す．

行列のエントリを変更するための次に述べる規則を用いて，システムの保護状態を管理する：

1. 領分 $x(i)$ が領分 $x(j)$ に対して制御属性を持っているならば，$A(x(j), y)$ から属性を取り除くことができる．
2. アクセスの属性 $A1$ が，行列のエントリ $A(x(i), y)$ に入っていて，かつコピー・フラッグが立っていれば，領分 $x(i)$ は $A(x(i), y)$ に $A1$ をコピーすることができる．
3. 領分 $x(i)$ がyに対する所有者のアクセスを持っているならば，$x(i)$ はアクセスの属性を，コピー・フラッグを立てたり立てなかったりして，$A(x(j), y)$ に送ることができる．

前述の規則は，アクセスの権利を取り消すための機会を与えていない．領分に取り消しの能力を与えるために，次のような規則を導入できる：

4. 領分 $x(i)$ がyに対する所有者アクセスを持っていて，$x(j)$ がyに対す

る保護されたアクセスを持っていない場合は，$x(i)$ は $A(x(j), y)$ からアクセスの属性を取り除くことができる．領分 $x(j)$ は，“保護”されていないようなアクセスの権利を何も受け付けなければ，取り消しに対して安全性を保障できる．

前述の方式は，主体を領分に関係づけたり主体を一つの領分から他の領分へ移すための機構を与えてはいない．新しい領分を生成することも扱っていないし，また保護状態の記述から表面的な対象物や領分を消滅することも扱っていない．しかしながら実際の実現においては，これらの機構を必ず含んでいなければならない．

7.4 保護の実現法

保護を実現する方式とは，保護の機構を実現するプログラムやデータ構造を集めることである．オペレーティング・システムにうまくのせることのできる実現方式には，幅の広い選択の余地がある．大抵の実現の問題の場合と同じように，通常，計算のオーバーヘッド，空間，柔軟性などの兼ね合いを決めるという問題がある．

保護の状態を表わすために行列を用いた場合，最も単純な実現方法は三つ組［領分，対象物，アクセスの属性］の記述を用いた表 T を与えることである．$A(x, y)$ の値が必要になったときはいつでも，この全域的な表を探索する．しかし，この実現方法は，通常，次のような理由により実用的なものとはいえない［Lampson, 1971］：

1. 表が非常に大きくなってしまうので，それをすべて主記憶の中に格納しておくことは非現実的である．重要な部分を主記憶に格納し，適切に保護しなければならない．
2. 対象物や領分の特殊な集団化を利用することが難しい．たとえば，公開用ファイルは各領分に対して表のエントリを持たなければならない．
3. ある領分がアクセスしたり所有したりするようなすべての対象物を知ることが，ときどき必要になる．そのような要求を効率よく処理できるように表 T を構成しなければならない．

第二の実現方法では，領分がアクセスするすべての対象物をまとめてリストにし，それをその領分につけておく．このリストは**資格リスト**（capability list）というが，これは対象物yに対する領分xのアクセス権をあらわす資格［$y, A(x, y)$］のエントリを持っている．もしも資格リストを格納するために使用できそうな読み取り専門の高速配列をハードウェアが提供してくるならば，うまくゆくであろう．しかし，異った**記憶キー**[†]（storage key）で保護しているような単一の記憶領域にすべての資格リストを保持すれば，同様の効果を得ることができる．高度な権利を持つ保護モニタだけが，この領域にアクセスすることが許される．

第三の実現法では，ある特定の対象物にアクセスするすべての領分をまとめてリストにし，それをその対象物につけておく．**アクセス制御リスト**（access control list）と呼ぶこのリストのエントリは，［$x, A(x, y)$］という形をしている．すなわち，それは対象物yに対するアクセスの権利を持っているような各々の領分に対して，アクセスの属性を定めている．各対象物に対しその対象物を生成した主体は，一つの手続きとアクセス制御リストとを提供する．その手続きは，その対象物にアクセスしようとする領分のチェックを責任を持って行なう．

第四の実現法では，ロック／キー機構において一意的なビット型を用いる．各領分においては，対象物のリストと共に各対象物に対するビット型を与えておく．そしてそのビット型（すなわち，キー）は，その対象物に対するアクセスの権利の識別名として働く．その対象物に付随して一意的なビット型（すなわち，ロック）のリストと，各ビット型に対するアクセスの権利が一緒に与えられている．一つの対象物にアクセスしようとする主体は，その対象物を保護している保護モニタにビット型（すなわち，キー）を提示する．モニタはその対象物に与えられているロックとそのビット型とを比較し，要求しているアクセスのモードにしたがってそのアクセスを認めたり拒否したりする．

キーが偽造できないものと仮定すれば，ロック／キー機構は有効に働き，柔

† **訳注**．記憶のある単位ごとに与えられるキー（たとえば4ビットで構成される）．プログラムは**保護キー**（protection key）を持って実行をし，一般に記憶キーと保護キーが一致しなければ，書き込みを許さない．

軟性を持ったものになる．資格をあらわすビット型は，それらの意味を考えずによそへまわしてしまうこともできる．また対象物の所有者はあるアクセスの権利を取り消したり変更したりしたいときは，対象物についているビット型（すなわち，ロック）のリストを変更するだけでよい．

対象物の名前を知り，またその対象物に対するアクセスの権利を知ると共に，主体はその対象物がどこに位置しているのかも知らなければならない．この結合のための情報は，通常，対象物をどのように保護しているかということとは独立に決まる．結合の路は，誰からでも要求があったときに，自由にかつチェックをせずに，システムによって与えることができる．別の方法として，アクセスするかもしれないような対象物に対する結合路に関する情報を領分が持っていることもできる．しかし，結合のための情報を保護の実現の構造の中に含ませておくことは，時々，便利なことがある．たとえば，資格リストを用いている場合，対象物に対する結合路は資格が書いてある領域に持たせることができる．システムの他の部分の構成に応じて，結合のための情報をどれだけ容易に組み入れることができるかということは，保護の方式を選択する際の問題になることもあるし，ならないこともある．

最後の3つの実現方法は，アクセスの正当性を局所的に調べることにより速度を上げることができるという利点を持つ．資格リストやロック／キー機構は，結合情報をコード化しておくためには都合のよいものである．しかしながら，ロック／キー機構は重なりをさけるためにビット型を充分長くしておかなければならないので，実現のためには少々多くの空間を必要とする．

第7.3節で導入したモデルを思い出そう．各領分xの権利は，対象物yからアクセス権Aに対する写像をあらわす関数$F(x, y)=fx(y)$で表現できる．ただし，$A=\{A(1), \cdots, A(s)\}$は，読む，書く，実行するなどというものである．これまで導入してきた実現方式は，関数fxがすべてのxに対して同じ形式をしているという意味で，非常に基本的なものであった．写像は異った領分や対象物に対しては異ったものではあったが，これらは本質的には，表引きやビット型の照合を用いることによってすべて実現されている．

fxの制限された形式を取り除くことによって，もっと凝った実現方式が可能となる．もっと効率のよい実現方式を行なうために，領分や対象物の構造を利

用することができる．たとえば保護の権利に基づいて，領分をきちんと順序づけていると仮定しよう．すなわち任意の2つの領分 D1 と D2 に対し，D1 は D2 のアクセス権をすべて持っているかあるいはその逆が成り立つとする．領分の順序に従って，各領分には番号が与えてある．一番少ない権利を持っている領分は1，次に少ない権利を持つ領分は2などとなっている．この方式のもとでは，領分の番号が要求した対象物の番号よりも大きいか等しい場合には，アクセスを許す．同様の方式は，同一の対象物に対する異ったアクセス権を区別するためにも利用できる．

もう一つの有益な方法は，対象物とそれらを管理する保護モニタとをグループ化することである．各々の資格（すなわち，データ構造）を3つの部分に分け，第一はモニタ，第二は対象物，第三はアクセス権への参照のために用いる．資格は大きくなってしまうけれども，資格が参照している対象物とアクセスの属性を高速に識別することができるという利点を持っている．

多くの異った保護の実現方式が理論的には可能であるけれども，それらの大部分のものは，実際は大量のオーバーヘッドを引きおこす．ときには，単純で適当なアルゴリズムを選択することが最善の方法になる．複雑な方式は概念的にはきれいであるけれども，それらは多くの好ましくない副次的な効果をしばしばもたらす．

7.5　資格の受け渡しと形式

任意の保護機構は，それがどんなに巧妙に作られていても，多くの特殊なインターフェイスを用いてシステムをずさんに構造化してあると，簡単に抜け穴を作られてしまう．不正なユーザは，存在しないものと仮定はしているが，システムが自由に認めているような隠されたアクセスの路を用いて，情報にアクセスする．壁や入口と似ているという議論に戻ると，その状況は壁に多数の穴があいているけれども，お客には入口を使うように頼んでいるようなものである．商業用のシステムの大部分では，このような症状を呈している．（もし，可能ならば）現存する保護のギャップを埋めるためには，実現方法だけでなく，基本設計をも大きく変更しなければならない．それゆえに，システムが安全になるためには，保護機構によって管理しているもの以外の実体の間には，アクセ

スの路がないようにシステムを設計しなければならない．

この節で示す例は，保護を実現しているロック／キー機構に変更を加えたものである [Sevcik ほか，1972]．この方式では，プロセスを一意的に領分と対応づけている．それゆえ，領分という独立の概念は必要ない．各プロセスに対して，アクセス権のリスト（すなわち，資格リスト）を与えている．リストの各エントリは，対象物に対するアクセス権を符号化した小さなデータ構造である．このデータ構造は，本質的には資格の抽象的な概念を符号化したものである．この節では，資格および資格リストは抽象的なアクセス権と，その特定の実現との両方を指すことにする．

保護機構に関して行なわなければならない基本的な決定の一つは，資格の大きさと形式である．それらは必要な情報をすべて含むことができるように充分大きくなければならないが，必要以上の記憶を使ってはいけない．可変長のブロックを扱うと問題があるので，可変の長さは実用的ではない．固定長という制限の内で，その形式を充分に柔軟にして，システムのすべての部分で生ずる保護の要求を扱うことができなければならない．

一つの方法は，資格が参照している保護モニタに応じて形式を変えることである [Sevcik ほか，1972]．資格は5つのフィールドを持つが，そのうち2つが固定長，3つが可変長である．資格全体の大きさは固定されている（64ビット）が，資格の内部のフィールドの位置は変化する．第一に，資格の型を指定する“型”フィールド（8ビット）がある．各資格の型は，単一の保護モニタ（すなわち，保護モニタはプロセスである）で管理しているので，型フィールドは資格を発行しているプロセスを一意的に識別する．第二に，“形式”フィールド（8ビット）があり，資格の残りの部分をどのように使っているのか，またこの資格を受け渡したり複写することができるかどうかということを指定している．また，そこでは資格を持っているプロセスが破壊されたとき，資格に何が生ずるのかもある程度示している．残りの48ビットは，3つの型のフィールドに分割する．**論理フィールド**は，そのビットを用いてアクセス権を表わす．論理フィールド内のビットは，ある管理の下で，消すことができる（したがって，権利を減らす）．**数値フィルド**は，通常，その資格を介してプロセスに割り当てる資源の量を示す数を持っている．プロセスは，資格によって与えられた

権利を単に減少させることができるだけである．したがって，数値フィールドを減少させるかまたは分割して新しい資格を作成することができる．資格を管理する保護モニタだけが，数値または論理フィールドの値を増加することができる．**自由フィールド**は保護モニタの自由に任し，必要などんな形式の情報も含んでいてよい．自由フィールドの意味を知っているのは保護モニタだけであるから，このフィールドを変更できるのは，資格の型によって指定されている保護モニタだけである．

資格は資格リストにのっているが，それはハードウェアの記憶キーを用いて高度に保護しなければならない．明らかに，プロセスが自分で資格リストにアクセスできるとすれば，すべての保護機構は破滅してしまう．保護のために，これらのリストは中央モニタによって格納できなければならない．そして，それが資格の管理を行なうものと考える．第2．3節で述べた核心が，この機能を行なってもよいし，あるいは独立のモニタを構成することもできる．すべての資格は，中央のモニタに対する特別なコマンドを使用して，作成し処理し受け渡す．

資格は，その型に応じた保護モニタによってのみ作成できる．保護モニタは，資格をある資格リストにのせることを要求する．中央モニタは，その保護モニタがこの権利を持っているかどうか，簡単にチェックすることができる．プロセスは，資格のコア・イメージを貰って自分の資格リスト上の資格を調べることができる．このような資格のイメージは，プロセスがアクセス権を偽造するためには役に立たない．プロセスが持っている権利は，中央モニタが持っているそのリスト上に格納されている資格によってのみ決まるからである．そのリストはハードウェアで保護しているので，こわすことは非常に難しい．

プロセスの間で資格を受け渡すことを，高度に管理することは重要である．明らかに，物理的な資格をプロセス間で受け渡すと，偽造が容易になる．この問題は，物理的なデータ構造のかわりに資格リスト内の資格の索引を受け渡すことにより解決する．資格を受け渡すために，プロセスは中央モニタに対しその一つの資格（たとえば，i番目のもの）を，他のプロセスのリストに移すことを要求する．形式フィールドの1ビットによって，その資格の受け渡しができるかどうか指示してあれば，受け渡しはもっとこまかく管理できる．中央モニタは，資格を移す前にこのビットをチェックできる．

たとえば，プロセスがファイル・システムが提供している特権的な処理を実行したいものとしよう．プロセスは，要求を出しその処理を行なうことが許されていることを証明するために，資格の索引を与える．ファイル・システムは，中央モニタに対しそのプロセスの資格のイメージを要求する．その資格が容認できるものであれば，ファイル・システムはその処理を完了する．

資格は，また保護以外の目的のために使用することもできる．ファイルを部分ファイルに分割し，異った部分の間に保護の壁を作り上げるような基本操作を提供する基本ファイル・システムを考えよう．さらに，すべてのファイルは２Ｋ語のブロックの整数倍の大きさであると仮定しよう．基本ファイル・システムは，一つの**論理フィールド**と３つの**自由フィールド**とを持っている“基本ファイル”という型の資格を用いる．第一の自由フィールドは，VOLUME といい，ボリュームの内部識別名を指す．残りの２つの自由フィールドは，START と RANGE であるが，ファイルの開始番地と長さとを指定している．論理フィールドは，そのファイルに対してどんな操作を許しているのかを示している．たとえば，読む，書く，実行するなどと指定する．

基本ファイル・システムは，資格を本質的にはファイル・ディスクリプタとして用いている．たとえば，SUBSET コマンド SUBSET (CAPAB INDEX, CAPAB SLOT, START, RANGE) を考えよう．CAPAB INDEX で示す資格によって識別されるファイルが指定する範囲内に START と RANGE とが入っていれば，基本ファイル・システムは START と RANGE で指定される部分ファイルに対し，新しい資格（すなわち，ファイル・ディスクリプタ）を作成する．新しい資格は，CAPAB SLOT で示す資格の欄に置く．

基本ファイル・システムにおいては，資格は START, RANGE, VOLUME で決まるファイルに対する結合情報を得るために用いるだけでなく，一意的な識別名としても用いる．したがって，保護のための資格とファイルの識別のためのファイル・ディスクリプタを実現するために，ただ一つのデータ構造を用いるだけで済む．

二番目の例として，入出力システムの一部を構成している機器マネージャに要求される読み／書き操作について考えてみよう．機器マネージャは，その操作の料金計算をしたいと思う．そこで，DEVICE MANAGER という型の資格

を発行するが，それは許可する入出力操作の回数を指定する大きな数値フィールドを持っている．この資格を，入出力を必要とするサブシステムに配る．サブシステムは，資格を得たときに料金をとられるが，こんどはそれが関係するプロセスに任意の方針にもとづいて自由に配ることができる．

機器の資格は，最終的には入出力を望んでいるユーザ・プロセスが獲得する．OPEN 処理の後では，機器マネージャは OPEN コマンドによってユーザ・プロセスが読み／書きの実行の許可を得たことを知っているので，ユーザ・プロセスは機器マネージャを通じて直接ファイルにアクセスすることができる．数値フィールドが正の値を持っている限りは，プロセスは入出力に対して“料金を払う”ことができるものと思われている．機器の資格は，入出力操作の数のカウントのために使用している．この場合，資格の数値フィールドに情報をコード化することにより，保護のためばかりでなく測定や料金計算のために資格を用いている．

良い保護機構を設計するのは難しい．複雑なシステムにおける他の問題の場合と同じように，論理的に完全な解は必ずしも適当なものではない．解は，また費用に見合った効果がなければならない．保護機構に付随するオーバーヘッドの量は，いくつかの応用にとっては重大なものとなる可能性がある．特に，すべてのチェックが逐語解釈的に行なわれる場合がそうである．保護機構が非常にわずらわしいものであるならば，それは避けられてしまい，したがって，効果のないものになってしまう．それゆえ，便利さとか保全の問題から離れて，保護機構の設計者は設計の費用を注意深く見積もらなければならない．

7.6 保　　全　　性

大規模なデータ・バンクに格納されている特別な情報へのアクセスを制御することに関する議論が，最近多くみられるようになった．問題は，3つの大きな分野に分類することができる [Gotlieb および Borodin, 1973]．**情報のプライバシー** (information privacy) は，個人による情報のアクセスを管理する法律や倫理の問題である．プライバシーは人間の属性である．人間はプライバシーに対する権利が何であるかを，集団としてまた個人として正確に決めなければならない．**情報の機密性** (information confidentiality) は，データに対するア

クセスの規則である．機密性はプライバシーの考慮は別にしても，保護すべき実際の世界のデータに対して適用する．**情報の保全性**（information security）は，機密性の決定を保証するような手段や機構を意味する．そのようなものとして，これらの機構は機械の中のデータの表現に対する機密性の要求を実現する．

たとえば，個人の過去の精神的および肉体的問題(すなわち，その人の病歴)のプライバシーについては，強い一般的な感覚が国中に普及するであろう．この一般的な感覚は，ある形式の法律として記述してもよいし，または法廷での決定を支配する方針として解釈してもよい．医学的データの機密化に関しては，特別の法律または政府の決定として問題点をとらえられるであろう．これらの法律や決定は，今度は医療レコードのアクセスにおける保全性を与えるように，医療情報システムの設計に対して影響を与えるであろう．

個人の人間として，誰でもプライバシーに関する市民としての関心を持っているが，しかし，計算機の専門家が持つ主な関心事は，情報の保全性の問題である．一般大衆はアクセスの権利の管理が困難であることに関して，警告を受けるべきである．さらに，システムやデータ・バンクにおいて，費用に見合った効果のある方法で機密性の実現ができるような技術を開発しなければならない．

本章で論じてきたように，保護は情報の性質は何も考慮しないで，オペレーティング・システムの中で情報へのアクセスの管理を処理している．このような機密は**内的**(internal)である．内部保護機構は．ラベル，ロック，キーなどを用いてシステム内の権利のある主体だけが，情報を持っている対象物にアクセスできるようにする．情報の保全性は，システムのある**外的**（external）な様相をも含んでおり，主として2つの点で内部保護とは異っている．第一に，計算機で処理しているシステムの一部よりも，むしろ情報システム全体を考えている．アクティブな要素は人間である．たとえば，ユーザの識別名は，内部保護のためには正しいものと仮定するが，保全性の実施においては，その正当性の検証は，第一に重要なことである．第二に，データの性質や内容がアクセスの権利を決定するのに重要な役割を果たす．保全性の機構は，その決定をファイルの内容に基づいて行ない，伝送中にデータのチェックを行なったり，フ

ァイルを処理しているプログラムの性質を確かめることを主張することができる．あきらかに，安全なシステムは内部的に保護していなければならない．しかし，それだけでは適当でない．この節では，内部保護に加えて保全性の問題を処理するような方法を論ずる．

次の例は，選択的なデータの機密性によって課せられる潜在的な保全性の要求を提示している［Conway ほか，1972］．大会社における社員の人事ファイルを考えてみよう．そのファイルは，名前，給与歴，現在の給与，評価，部門，病歴，社会保障番号のデータ項目から成り立っている．個人によるこのファイルへのアクセスに関して，ある種の機密性の決定の例を次にリストアップする：

1. ユーザはこのファイルに対して，完全にアクセスできる．
2. ユーザはこのファイルに対して，なにもアクセスできない．
3. ユーザはこのファイルの任意の部分を見てもよいが，その内容を変更してはならない．
4. ユーザはちょうど一つのレコードを見てもよく，またそのレコードのあるフィールドだけ変更してもよい．
5. ユーザは名前フィールドと病歴フィールドだけを見てよく，病歴フィールドだけ変更してもよい．
6. ユーザは，各レコードの“お金”に関する部分だけを変更してもよい．しかし，一日のうちある決った時間に，決った端末からだけ行なう．
7. ユーザは，現在の給与項目がある量以下であるような“お金”に関するレコードだけを見たり，修正したりしてよい．
8. ユーザは，“お金”に関する情報を，集団として見ることができるが，個人レコードのものは見てはならない．
9. ユーザは，評価の項目について見ることができるが，それもある部門についてのみである．

前節に述べた保護機構を用いれば，上述のリストの最初の四つの要求は非常に簡単に処理することができる．残りの要求も，それを用いて概念的に扱うことはできるが，かなり効率が悪い．さらに，内部機構によっては処理できない

ような問題として，次のようなものがある：

どのようにすれば，個人の識別が保証できるか？
盗聴に対して，どんな手段がとれるか？
ディスクやテープがオフ・ラインになっているとき，どんな管理が実行できるか？
システムが，ハードウェアやソフトウェアの誤りのために正しく働かないとき，どんなことが起るか？

よい保全性の環境では，これらの問題すべてに対して解を与えようとすべきである．“破壊不可能”なシステム（商業用のシステムである必要はない）を設計するのは，ほとんど不可能である．保全性の機構を避けて通るのは，非常に困難で非常に高いものにつくように設計しておくことは重要である．保全性の問題は，完全なシステムを頭に置いて考えなければならない．保全性の破壊は，保護のチェインの一番弱いところから生じ易い．そこは必ずしも最も複雑で技術的にも進んだ処理を施してあるわけではない．システムはその最も弱い点と同じ程度に安全なだけである．

保全システムを破壊する最も直接的な方法は，特権的なユーザの信頼を利用して行なうことである．この問題はよく知られており，それを押さえる方法は手で処理するデータ・システムで提供されているものと似ている．人物の調査を行ない，保全性の破壊に対しては高い罰金を課し，重要な情報にはラベルをはって事故による秘密漏れを最小にする．銀行でよく使用する方法は，対を組ませて許可を与える方法である．重要な操作を行なうためには二人の人間が必要であるから，悪いことをしようとすると二人の意見が一致しなくてはならない．その拡張として，個人個人の機能を分離してしまうことを考える．たとえば，オペレータがシステム・プログラマも兼ねていると，そうでない人に比べて，ずっと容易に保全性を破壊することができる．最後に，保全のための番人がいることを単に示すだけでは必ずしも役には立たない．有効に働くためには，番人は保全システムとその中における自分の役割とをよく理解して，それを実行しなくてはならない．

一般的な規則として，特権的なアクセスとシステムの操作に関する情報とは，

両者とも出来るだけ少数の人にだけ教えておくべきである．“知らなければならない”ということは，第一の基準になる．

許可していない操作に対する保護をするためには，伝統的には，ユーザを**合い言葉**（password）によって識別してきた．しかし，この方法はいつでも適当であるわけではない．もっと多くの保護を与える精巧な機構も可能である．たとえば，ユーザがログ・アウトするときに，一意的な番号（すなわち，一時的合い言葉）を与えることができる．その人は，次にログ・インするときに，この番号を合い言葉として用いる．もしも，誰かが合い言葉を作り出して，しばらくの間，その人の名前を偽っていれば，その人の合い言葉が不正なものになってしまうので，自分の環境が破壊されたことを，少なくとも検知できるであろう．もう一つの方式は，算術的変換を用いる方法である．ログ・インの際にユーザが自分を識別するとき，システムは擬似乱数を与える．ユーザは単純な変換 $T(x)$ を行ない，その結果を送り返す．システムはその変換を高度に安全な領域に格納しておき，ユーザの識別名のチェックを行なう．x と $T(x)$ とは，T を識別するために情報をほとんど与えないから，この方式は通信線に対する盗聴に対しても高度に安全である．

一時的な合い言葉を用いても，正当なユーザの通信線に端末をつけるような侵入者に対しては保護ができない．そのかわりに，端末または中央処理装置のハードウェアで作り出した一意的な数字を用いれば，メッセージを識別できる．メッセージの源を，今度はチェックできる．ユーザの識別の問題は，また端末や通信線の物理的な位置の保全性とも関係してくる．

合い言葉を用いた保護機構は，侵入者が資源と忍耐強さとを充分持っていれば，どんなものでも侵害されてしまう．したがって，監視を行なえば，システムが侵害の企てを見つけ罰金を課すことが可能となる．ファイルの処理に対する周期的な報告は，性能評価やシステムの調整に役立つだけでなく，誤った使用や悪だくみの指摘をするのにも役に立つ．強制的な監視がないときには許可されていないアクセスをたくさん企てることにより，システム内に多くのオーバーヘッドを生み出す可能性がある．各アクセスは拒否されるけれども，それにも拘らずある注意を払う必要が生じてくる．

操作の会計日誌を用いれば，保全性の侵害をオフ・ラインで検出することが

できる．会計日誌は解釈が難しいけれども，それは会計の履歴を示し，保全性の破壊の証拠を与えてくれる．そのように用いれば，それは不正行為の防止者として働く．

ハード・コピー，取りはずし可能ディスク，テープなどにあるデータは，オフ・ラインで適当に保護しなければならない．番人，地下倉庫あるいは他の手動のシステムは，保全性の手段を与えている．また，所有者にとっては役に立たないが保全性の破壊を引きおこすようなデータ，たとえば，古いテープ，いらない紙，古いコア・イメージなどに対しては注意を払わなければならない．主記憶やテープは，使用後“汚れた”ままにしておくのが習慣である．それらはその情報を消去するために，乱数を上書きしてきれいにしておくべきである．ある種のバックアップの機能が保全のためには必要である場合でも，必要のないコピーをふやすことは避けるように注意しなければならない．

内部保護を議論する際には，対象物に含まれるデータに対してはほとんど注意を払わなかった．良い保全システムにおいては，アクセスの注意深い監視を保証するために，データの変換とかデータやプログラムの性質の調査にときには戻らなければならない．

情報を隠すために，重要なデータに対し逆変換可能な符号化を行なうことができる．これにより，伝送中の盗聴，データ・ファイルに対する許可のないアクセス等に対し保護が可能となる．保全性をやっと維持しているような現存のシステムのサブシステムとして，高度な保全性のあるシステムをつけることが必要なときに，これは特に役に立つ．文字列の代入，文字の位置変換およびキー文字の加算の3つの方法がよく使われるが，これによりコードの破壊に対する複雑性を増すことができる［Hoffman, 1969］．

ユーザやプロセスの権利に従って，ファイルのレベルでデータのアクセスを管理するだけでなく，アクセスされるデータに従って，レコードのレベルでアクセスの管理をすることが望ましいこともある．この方法を用いれば，特に情報を共有するような環境において，異った資格のコピーをふやすことを避けることができる．たとえば，前に述べた人事ファイルにおいて，給与のフィールドのデータの値を全然考慮せずに，15000 ドル以上の給与の項目を見ることをユーザにを制限することはできない．15000 ドル以下の給与を持つレコードに

対しては，別の資格を与える．別の給与レベルが望ましいときは，別の資格が必要になるだろう．別の方法として，資格によってあらわすような特別のアクセスに付随した保護モニター内の手続きを呼び出して，データに依存するアクセスの決定を逐語解釈的に行なうことができる．資格に関する手続きは，非常に一般的な形をした保護を与えてくれる．しかしながら，操作が逐語解釈的に行なわれると，かなりの量のオーバーヘッドを生ずるので，必要なときにだけしか使用してはならない．

保全性の機構は，プログラムのある性質の検証をも含んでいる．そのような機構は，プログラムの検証という一般的分野に属する．たとえば，処理するデータについての統計的な情報を得ることができないように，プログラムを作るべき場合を考えてみよう．プログラムの構造を解析することによって，それが正しく振る舞うということを証明してくれなければ，保全の機構はそのような条件を強制するための方法を何も持っていない．ときには，コンパイル時にコンパイラが作成したアクセスの要求を調べるというのも，良い方法となることもある．そのような場合，不正なアクセスを誤って提供しないということについて，コンパイラを信用しなければならない（すなわち，正しいと証明されなければならない）．さらに，大抵のシステムにおいて，保全性の機構はソフトウェアで実現している．保全性を提供している手続きが，保護され，検証されていることは重要なことである．

保全性は，最近，非常に重要になってきた．人々は大きな計算機を用いた情報システムによってアクセスできるように提供されている情報の量に，だんだん気がつきはじめている．保全性の破壊は，以前は確かに可能であったが，しかし，手動システムにおいては，非常に非現実的であった．計算機を用いたシステムでは，データに対して簡単にアクセスできる．それゆえ，情報を得ることの非現実性や複雑性にだけ頼っていないで，むしろ，許可していないアクセスを許さないことを実現すべきである．

7.7 おわりに

本章では，内部保護をもっぱら扱い，それを詳しく論じてきた．いろいろの機構が，現在までに提案されてきた．適当な保護機構を作ることは，設計者の

主要な目的の一つとすべきものである．保全性の問題は，簡単に概略を述べたが，本質的には非常に技術的なものとはいえないけれども，非常に重要なことである．

計算機科学者は，その活動を神秘主義の雲の後に隠しているという批判を受けている．システムの保全性の機構は，特に，微妙な問題である．システム設計者，プログラマ，オペレータは，保全性の破壊を利用することができる．法廷に送付された件数から判断すると，ときどき，それが行なわれていることは明らかである．したがって，管理の側から見ればうまく働くと主張していても，不明瞭であるような機構を極度に警戒する．保全性の機構はうまく働くだけでなく，技術者でない人々，たとえば，銀行の経営者のような人々に，容易に理解できるものでなければならない．

演 習 問 題

7.1 記憶が，保護している部分と，保護していない部分の２つの部分に，動的に分割されると仮定しよう．各語に対して，１個の保護ビットを設けると，語の状態が保護されているかいないか示すことができる．また，プロセスは特権的および非特権的モードで実行できる．特権モードでは，すべての記憶の語はアクセス可能であり，各語の保護ビットを変更することもできる．非特権的モードでは，保護していない記憶だけがアクセス可能であり，保護ビットは変更できない．任意個のプロセスを有するシステムにおいて，各プロセスが自分自身の記憶領域を保護できるようなソフトウェアの方法を設計せよ．

7.2 システム内で，各プロセスXに数xを，また，対象物Yに数yを割り当てるものと仮定する．そして，保護を次のように実現する．“プロセスXは，$x>y$ が成り立つとき，かつそのときだけ，対象物Yにアクセスできる”．この実現方式はプロセスのアクセス権に対して，どんな構造をもたらすか．これを，MULTICS のリング保護[†] [Schroeder および Saltzer,

† 訳注．リング状の領分を，ハードウェアの助けを借りて実現する．内側のリングほど強いアクセス権を与えあり，OS の核などを置き，外側には，ユーザ・プログラムなどを置く．一つのプロセスの実行過程において，動的にリング間の移動も行なう．

1971] と比較せよ．木構造を実現するような方法を考えよ．そのような方式の動的な性質を分析せよ．

7.3 小型のミニコンピュータの場合のように，主記憶にハードウェアの保護がないと仮定せよ．うまく保護されたシステムに到達するまでのステップの系列を，簡単に述べよ．二次記憶の情報を保護する方法は，何かあるか．

7.4 一つの言語から成るシステムを考えよう．ユーザが，お互に保護されることを保証するために，どんな特徴がコンパイラに組み込めるか．

7.5 資格の処理を効率よく行なうために，どんなハードウェアの特徴が必要か．資格を使用した機械の設計の概略を述べよ．与えられた機能は，保護以外の目的で使用することができるか．

7.6 システム内には，多かれ少なかれ公開されている対象物が，いくつかある．すなわち，ほとんどの人からは読まれるが，少数のプロセスからだけ書き込まれるものである．不必要なオーバーヘッドをもたらさずに，これらの対象物を適当に保護する方法を考案せよ．

7.7 保護における主体が，手続きであるようなシステムを考えよう．領分を，手続きに与える方法を考えよ．一つの手続きが，他の手続きを呼び出そうとしたとき，呼び出す手続きは呼び出された手続きが必要な対象物にアクセスできるように，領分を受け渡さなければならない．手続きの呼び出しの引数として，どうやって領分を渡すことができるか．

7.8 資格を取り消す能力が重要であるような例を見つけよ．資格の取り消しが，重大な問題を生ずるような別の例を見つけよ．

7.9 システム内の異った対象物を保護するために適用できるような可能なアクセス権（たとえば，読み込みのみ，読み／書きなど）のリストを作れ．これらの権利を，資格の論理フィールドに符号化してあるものと仮定せよ．論理フィールドの望ましい長さは，どれだけか．答の正当性を示せ．

7.10 所得税を計算する手続きを考えよ．この手続きは，収入が15000ドル以

下の課税所得を持つユーザだけが呼ぶものとする．課税所得は，それ自身，税金計算の中間結果である．さらに，この手続きは，与えられた個人の税金の状況に関する情報を保持していてはならないものとする．そのような環境を実現する機構の概略を述べよ．

7.11 〈主体，対象物，アクセス権〉という形式のエントリを持つ基本的な保護の実現方式を考えよ．特別な主体や対象物のエントリに，非常に速くアクセスする方法の概略を述べよ．その実現のためには，どんなハードウェアの機能が役に立つか．

7.12 第7.4節に概説した4つの基本的実現法の各々に必要な記憶を分析せよ．

7.13 ユーザの識別をする3つの良い方法を考案せよ．各々に対して，その方法が働かないような状況を見つけよ．システム内の特定の主体が，うその名前を使えば，保護システムは破壊してしまう．たとえば，プロセスが，SUE の**ファシリティ**（facility）（第10章参照）をかたることができれば，そのファシリティに付随している資格を侵害することができる．少なくとも，システム・プロセスの間では正しい識別の検証が保証できるようなステップの系列について，概略を述べよ．

7.14 任意の領分や対象物の集合に対し，資格リスト，アクセス制御リストおよび表[†] のエントリの数が同じであることを示せ．

7.15 ハードウェアの記憶保護の機能の概括をせよ．ベース・リミット・レジスタ，保護キー，Honeywell-MULTICS の保護リング，BCC-M1の資格などの利点の評価をせよ．

7.16 異なったいくつかの大型のオペレーティング・システム，たとえば，Burroughs MCP，IBM OS/360，CDC Scope などで使用できる保護機構について，小論文を書け．

7.17 社会における個人のプライバシーに対する計算機の影響について，小論文を書け．

---

† **訳注**．第7.4節の実現法で述べた表T．

第 8 章

設　　　計

8.1 はじめに

オペレーティング・システムは，かなり大規模で複雑なソフトウェア・システムである．それゆえ，小さなプログラムに対して用いるものと同じような形式化されていない組織的環境の下で，それらを作り出すことはできない．時間と資源に関してある限られた制限の下で，オペレーティング・システムを作り出すという特別な目的をもったプロジェクトを打ちたてなければならない．重要な作業をする時の第一歩として，提案書を書かなければならない．これは，プロジェクトの資金源とは無関係に正しいことである．提案書は外界に対して，プロジェクトの目的を示すためだけでなく，少なくとも内部のドキュメンテーションのためにも役立つのである．

提案書には，システムの一般性，効率，信頼性などの見地からみたプロジェクトの目的が書かれていなければならない．関連のある情報の文献については，概括していなければならない．他のプロジェクトにおける経験も，可能であるものはどれでも利用すべきである．いくつかの目的が矛盾をきたすならば，それらの相対的な優先度を考えなければならない．製品のテストの方法について，宣言した目的と許容の基準との見地から，概略を述べていなければならない．主要な要素の機能とか全体の内部構造のようなシステムの仕様を与えなければならない．提案書にはプロジェクトの完成に対する時間と費用の推定がなければならない．人的な要求は組織内の現状と外部からの支援の可能性との両者を現実的に考えて，明確に述べなければならない．進行状況を把握するために，頻繁にチェックポイントを行なうようなスケジュールを用意しなければならな

い．またチームの構成の概略も，述べなければならない．市場の戦略のような技術的でない理由によって，プロジェクトの開始のときの決定がすでに到達されているとしても，プロジェクトの完成によって得られる利益を提案書の中で強調しなければならない．個人的な理由では，人々は何か重要で刺激的なものに参加したがるものである．

プロジェクトの中心となる人々は，提案書の作製に参加すべきである．それらの人達が，最初からそのプロジェクトに対し個人的な興味を持っていることが重要である．その提案書は，プロジェクトの要員と組織の中には広く回覧しなければならない．興味のある人あるいは重要な地位にある人達には，内容についてのコメントをきくべきである．外部からの資金を用いる場合は，提案書こそ資金を獲得するための実際の役割を果すのである．

提案書が書かれ受け付けられた後では，プロジェクトは公式的に開始したことになる．プロジェクトの要員は，それから実際にやってみたい一般的な設計の方法を選択すべきである．ドキュメンテーションの道具，デバッギングの道具，プログラミング言語などのような設計や製作のための道具は，どこからか持ってくるか作製するかしなければならない．このような準備が完了してしまえば，実際の設計の仕事が開始できる．

この章では，いくつかの異った設計方式の概略を紹介する．それらはどれも絶対的に完全なもの，誤りのないものというわけではない．読者は自分の判断により，自分の生産力，環境，また最後にかなり重要ではあるが自分の性格に一番適しているような設計の方法論を採用すべきである．

8.2 設計の方法論

設計とは創造的な過程である．オペレーティング・システムの設計において従わなければならないような，はっきりした規則は何もない．われわれは自分の経験あるいは他人の経験から学び，どのような技法がよいシステムのために役に立つのか，どのような問題を避けなければならないのかを決定すべきである．しかし，従うことのできる一般的な指標と全体的な方法が，いくつか存在する．

ソフトウェアに対する設計の方法論は，急速に発展している．ソフトウェアの設計原理は，ある程度まで他の分野での設計原理と似ている．単に術語を翻

訳するだけで，他の分野に対する設計の方法論がソフトウェアの設計に対しても適用できる（たとえば，市街の設計 [Alexander, 1964]）．しかし，計算機システムの設計に，特に適用できるような原理がある．この節では，オペレーティング・システムの設計や実現のために用いてきたいくつかの方法を調べてみよう．

定義に従えば，設計の過程とは，設計される製品の形や内容に関して決定を次々と行なうものであると考えられる．設計法は設計者の決定を命令するのに役に立たなくてはならない．設計者の決定が適当な前後関係の中で，最適な時期に行なわれたということを知れば，それらの決定が正しいものであることを信ずる気持が強まる．さらに設計法によって，設計者が製品の特定の様態や部分について，独立して集中できることが可能でなければならない．この点はオペレーティング・システムでは，特に重要である．システムが複雑であるので，ある人がすべての詳細な点を一時に理解するということは不可能だからである．それにも拘わらず，設計者は設計中にシステムが成長するにつれて，その全体の性質を完全に理解していなければならない．

設計者が，システムを分割しその複雑さに打ち勝つために役に立つ最も能力のある道具は，抽象化という概念である．**抽象化** (abstraction) とは，システムまたはその一部分を完全に詳細には決めずに記述するということである．抽象化によって，設計者はシステムを巨視的に見ることができる．そのようにすれば，システムのすべての詳細が示されたときには非常に困難であるようなシステムの全体に及ぶ関係や主要な要素の性質に対する見通しを得ることができる．

すべての記述は，記述される対象が物理的な原型でない限り，抽象的なものでなければならない．たとえば，計算機システムの記述は，基本的なハードウェアやソフトウェアの特徴以上に詳細なことは，恐らく何も規定しないであろう．エレクトロニクスの構造は，記述からは隠されている．この見地からすると，物理的な対象物の記述はそれが対象物そのものでない限り，完全なものとはならない．

抽象化は何も新しい考え方ではない．それは，長い間，数学の世界で用いられてきた．学生達が学んだ第一の抽象化は，代数において数の性質をそれらが

表わしている物理的なものの量から分離したことである．たとえば，数5は，5つのりんごの入ったバスケットを抽象化したものである．

特定のシステムの**唯一の**抽象化というものは，存在しない．すべてのシステムに対しては，多種類の抽象化が可能である．各抽象化はシステムがどのように見えるかとか，どんなことをするのかということに関する異った見通しを，見るものに与えてくれるのである．しかしながら，システムの抽象化はそれぞれ密接に関係づけることができる．たとえば，一つの抽象化$a(2)$は，もう一つの抽象化$a(1)$の**精錬**（refinement）されたものとすることができる．$a(2)$が$a(1)$を精錬したものであれば，$a(2)$は$a(1)$が記述したものを，すべて記述しているし，また$a(2)$の詳細さは少なくとも$a(1)$のそれと，いつも同じ程度である．すなわち，$a(2)$は$a(1)$の記述を，一様に詳細化したものである．

$a(1), \cdots, a(n)$は，各iに対し$a(i+1)$がその前の抽象化$a(i)$の精錬したものであるような抽象化の系列であるとしよう．各抽象化$a(i)$は完全であり，必ずしも詳細ではないけれども，システム全体の記述になっている．iが増加するにつれて，記述の中の詳細の度合いは深まる．抽象化がこのように入れ子になっているときには，**抽象化のレベル**（level of abstraction）すなわち記述における詳細の度合いに言及することは意味がある．抽象化のレベルが高いときは，細かいことはほとんど含まれていない（すなわち，高度に抽象化されている）．抽象化のレベルが低いと，非常に細かいことがたくさん含まれている．抽象化のレベルは相対的な概念である．抽象化のレベルが高いというときには，それよりももっと詳細な何か他の記述と比較しなければ，意味がない．しかしながら，計算機システムの記述はかなり明確に詳細度のレベルに分けることができる．ある人が高度なレベルのシステムの記述というときには，通常，システムの構造については何も詳しい指定はせずに，システムの全体としての特性を述べていると考えられる．低いレベルの記述は，一般に，実際のコードとハードウェアの構造とを扱っているものと考える．したがって相対的にどの位高いレベルとか低いレベルとか明瞭にせずに，高レベルのシステム記述とか低レベルのものとか述べることが，非常によく行なわれる．ハードウェアは低く，ソフトウェアは高いという意味であることは理解されている．

オペレーティング・システムの一つのモジュールとしてのファイル・システ

ムを考えよう．ファイル・システムの性質を記述する一つの方法は，それが処理するすべてのコマンドの名前と，各コマンドによって実行される機能のリストを作ることである．これはファイル・システムの，かなり高レベルで巨視的な抽象化である．そのようにすれば，ファイル・システムは何をするのか，オペレーティング・システムの他の部分がそれをどのように使うことができるのかということを，基本的に理解することができる．つぎに，ファイル・システムを作り上げている相互に連絡し合っているプロセスの集合を考えてみよう．これらのプロセスは，一緒になってファイル・システムの全体的な内部構造を記述するもう一つの抽象化を作り上げる．ファイル・システムの各プロセスは，データ構造の集合および手続きの集合から成る独立な単位と見做すことができる．各プロセスに現われるこれらの手続きやデータ構造の非形式的な記述は，やはりファイル・システムの低レベルの抽象化を与えている．各手続きやデータ構造は，ALGOL のようなプログラミング言語によって，さらに詳しく指定することができる．これは，やはりシステムのもう一つの抽象化を与えている．この最後の抽象化は，機械的に機械語に変換できる形式をしていると仮定してもよいという点で，少々異なったものである．この時点において，抽象化はその抽象的な性格を失い始めるのである．これはファイル・システムの性質の抽象化というよりも，むしろその具体的な表現と考えられる．

二番目の主要な設計概念は，抽象化とも密接に関係しているが，仮想機械ということである．**仮想機械**（virtual machine）とは，システムの操作やシステムの一部分を記述する基本的な操作の集合である．その機械は，必ずしもハードウェアで実現しているわけではないので，“仮想”と呼ぶ．たとえば，ALGOL 言語は仮想機械を記述する．ALGOL 仮想機械は，ALGOL 言語を機械語に写像するような ALGOL コンパイラによって，通常実現する．機械語も，やはり仮想機械を記述している．しかし，この仮想機械は実機械として実現される．

抽象化のレベルの場合と同様に仮想機械のレベルを考えることができる．仮想機械のレベルが低ければ低いほど，ハードウェアに近くなる．ALGOL 仮想機械は機械語仮想機械に比べて，高いレベルにある．仮想機械は，典型的には他の仮想機械の上で動いているいくつかのソフトウェアによって，実現してい

る．機械語のハードウェアの上で動いている ALGOL コンパイラは，ALGOL 言語を実現している．ALGOL 言語で記述し，ALGOL 仮想機械の上で実現しているファイル・システムは，ファイル・システムのコマンドによって特徴づけられるような仮想機械を実現していることになる．システムのユーザが ALGOL コンパイラとファイル・システムだけを見ているとすれば，ALGOL の命令とファイルのコマンドを解釈するような仮想機械の上で実行していることになる．もしも，そのユーザが統計解析のためのプログラム・パッケージを書いたとすれば，そのパッケージのユーザは統計解析を実行するような仮想機械を見ていることになる．それゆえ，ソフトウェアの各層は新しい（高度な）レベルの仮想機械を実現していることになる．

仮想機械のレベルと抽象化のレベルとは，関連のある概念である．システムの抽象化とは，ある言語でそのシステムの記述をすることである．その言語で許している詳細の度合いにより，その言語で記述した抽象化のレベルが決まる．たとえば，システムを ALGOL とアセンブリ言語とで記述していると仮定しよう．ALGOL による記述の方が詳細の度合いが小さいので，あきらかに前者の記述の方が，後者のものよりも高いレベルにある．ALGOL とアセンブリ言語とは，また仮想機械を記述している．システムを記述するために用いる言語は，どれでも仮想機械すなわちその言語を実行できるような機械を規定している．これは，ちょうど言語とはあるレベルの詳細さでシステムを記述するための道具であると言っていることになる．高レベルの仮想機械を規定する言語は，高レベルの抽象化に対応するシステムの記述をつくり出すために用いることができる．高レベルの仮想機械の究極的なものは，自然言語の記法を処理するものであろう．英語で記述したシステムが，実際に動いているシステムになるためには，英語の記法の仮想機械の上で実行できなければならない．

システムの基本的なモジュールまたは**ビルディング・ブロック** (building block) が何であるのかを問うのは，意味のないことである．それはシステムを見ている抽象化のレベルに，大いに左右される．高レベルの記述は，低レベルの記述とは異なったビルディング・ブロックを用いる．システムを記述するために用いる言語によって，ビルディング・ブロックは，手続き，プロセス，関数モジュール，サブシステムなどになる．

抽象化，精錬化，仮想機械という道具を与えられて，設計者は全体の構造を見失うことなく，システムの特定の部分に注意を集中することが出来易くなる．以下では，これらの概念を用いて，設計上の決定を命令するいろいろの方法を概括してみよう．

8.2.1　構造的設計

構造的設計（structured design）とは，通常システムの異なった部分を分離するために，構造を用いることを意味する．そうすると，各部分は独立に設計できる．オペレーティング・システムの最も伝統的な設計方法の一つは，システムの主要なモジュールを独立に設計することである．これは，時には**モジュール－インターフェイス**（module-interface）**方式**という．実在のシステムを調べてみると，オペレーティング・システムは，入出力システム，処理装置の時間のスケジューラ，記憶マネージャ，ファイル・システムなどに分割されていることがわかる．各モジュールを分離し仕様を決める（通常，日常語で）．そして，他のモジュールに対するインターフェイスの概略が決まる．モジュールとそれらの相互結合は，システムの一つの高レベルの抽象化を作り出す．この時点において，モジュールはその先の設計や実現に対して，独立なものとして考える．同一の手続きをシステムの単一のモジュールの内部でも繰り返す．すなわち，一つのモジュールに対する作業をさらに分割して，個人に対するプログラミングの割り当てを行なう．すべてのモジュールを設計し実現すれば，事前に決められたインターフェイスに従って一つに結合する．

モジュール－インターフェイス方式においては，正しいインターフェイスを保証するということが重大な問題となる．モジュールやそのインターフェイスは，通常，非常に非形式的で不正確な方法で，仕様を決める．そのために，個々のモジュールは仕様に従って書いてあっても，操作の誤っている場合をしばしば仮定してある．モジュールを結合したとき，それらはうまく接合しない．モジュール間で共有するデータ項目に対して考えた論理的構造や内部表現は，その項目を用いる各モジュールに対して，異っているかもしれない．データ項目の仕様に対する矛盾を正すことは，モジュールをコード化した後では極めて困難である．時には，2つのモジュール間のインターフェイスを正すためだけ

の目的で，特別なプログラムを作らなければならないことさえある．自動的なインターフェイスの接合に対して，特別なシステムが提案されている［Weissman および Stacey, 1973］.

原理的には，インターフェイスを完全に決めていれば，何の問題も生じない．残念なことに，最初の全体の設計や計画はプログラミング言語のレベルの詳細にまで及んでいない．その結果，最初の時点で経験の深い設計者によってなされるべき重要な決定が，モジュールの中に押し戻されてくる．最後の時点では，プログラマはシステムの非常に限られた狭い視野に立って，大域的な意味を持つ決定をなすことになる．適当でない時期に，あまり資格のない人によって，決定がなされてしまうかもしれない．たとえば，オペレータ・コンソール・マネージャのような重要なモジュールの仕様が完全に決まっていないときは，各モジュールはオペレータ・コンソールとの通信の仕方について，異った仮定をするかもしれない．そのような誤りを修正することは，非常に難しいであろう．これは決定が正しく伝わらなかった例である．良い設計方法は，設計者の決定が正しく伝わることを助けるようなものでなければならない．

モジュールを適当に識別したり，分離したりするのは，困難な仕事である．システム内の密接に関係のある部分は，同一のモジュール内に存在すべきである．システムの2つの部分の結合の度合は，一つの部分が他の部分について必要とする仮定の数であるという提案がなされた［Parnas, 1971］．隣りのモジュールが働いている環境に関して，非常に多いかまたは非常に少ない情報しかない場合は，モジュールの設計がまずいという結果になる可能性がある．たとえば，データ構造を処理しているモジュールAを考えよう．隣りのモジュールBはAとの通信のための正確な**プロトコル**（protocol）を知らなければならない．しかし，それはAの局所的なデータ構造に関する正確な詳細を知らなくてもよい． Parnas は他のモジュールの内部仕様の情報を非公式的に用いると，一つのモジュールの設計において害になることがあり得ると述べている［Parnas, 1971］．たとえば，前述の例でモジュールBがAの局所的なデータ構造に関する情報を用いたと仮定しよう．この時点において，Aがその構成を変える必要が生じたとき，モジュールBは非常に困ったことになりかねない．モジュールBの仮定は，システム全体が破滅するまでわからないかもしれない．

モジュール－インターフェイスの方式は，構造的設計法の一つのやり方である．モジュールとそのインターフェイスの仕様が，各モジュールおよびシステム全体の設計に対する構造的な基礎を与える．比較のために，橋の設計を考えてみよう．設計者は橋というものは橋台，橋脚，支えなどの異った部品から成り立つということを知っている．それゆえ，橋の異った部品を分離し，それらを独立に設計することができる．さらに設計者はこれらの異った部品を，どのように接続するかを指定しなければならない．部品の結合において，何か問題があるとすれば，たとえ各部品は個々の仕様に合致していたとしても，橋を完成することはほとんど不可能である．部品を接合するための特別な解決法を，見つけ出さなければならないだろう．

ソフトウェアの設計者は，システムの非常にあらっぽい不完全な骨格だけを考えて出発し，個々のモジュールの詳細について早めに考えてしまうことがある．その結果，システム全体に影響を及ぼすような決定を，適当な前後関係でないときに，その結果を完全に理解することもなく下してしまう．あわてて実現すると不安定なソフトウェア・システムになってしまい，膨大な保守の努力をしばしば必要とする．残念ながら，このような状況は，警告を受けていてもおきてしまう．

次の節では，構造的設計法と抽象化の原理に基づいた２つの方法を述べよう．本質的には，それらはモジュール－インターフェイス方式と似ている．しかし，それらはモジュールの分割の仕方と結合の仕方とを，厳密に制限している．この構成は，設計の決定をある順序に従って行なうことを強制している．厳密に順序付けることには欠点はあるけれども，決定を何らかの順序に従って行なうことは，何も順序づけずに行なうよりもよい結果をもたらすことが多い．設計の過程に順序を課すという原理を追加すれば，決定が遅すぎてしまうということはなくなる．

第１章に述べたように，オペレーティング・システムの主な目的はユーザに対し，(1) 柔軟性のある機能，(2) システムの資源の効率の良い共有を提供することである．それゆえ，ユーザが実際に見るものは基本的ハードウェアよりもずっとオペレーティング・システムに似た性格を持った仮想機械である．オペレーティング・システムはユーザの仮想機械とハードウェアの機械との間に橋

をかけているのである．オペレーティング・システムをこの様に考えると，その設計者は，少なくとも2つの方法で進むことができる．すなわち，ハードウェアからユーザの仮想機械へ上向きに進むか，またはユーザの仮想機械から実在するハードウェアへ下向きに進む方法である．前にも述べたように，もっと柔軟なユーザ向きの仮想機械は，伝統的には基本的な機械よりも“高度な”ものと考える．したがって，この2つの設計方法は**上昇**(bottom-up)と**下降**(top-down)という．

8.2.2 上昇設計法

上昇設計法においては，設計者は基本ハードウェアから出発する．裸のままのハードウェアは，問題解決のために用いるのには，かなり不便な機械である．それゆえ，設計者はソフトウェアの層をつけ加える．ソフトウェアはその下にあるハードウェアと相まって，新しい仮想機械を定義するコマンドの集合を提供する．次のステップでは，もう一つ必要な特徴を分離し，ソフトウェアの層をつけ加えると，さらに便利な仮想機械が得られる．ソフトウェアの層を次々とつけ加えるが，各層は最終的にはユーザが望んでいる仮想機械に到達するまで，いくつかの望ましい機能を実現してゆく．これが基本的な上昇設計法である．

一つのソフトウェアの層は，それを支援するソフトウェアとハードウェアの層とが一緒になって仮想機械を定義する．そのソフトウェアの層が提供する機能のほかに，新しい仮想機械は下に存在している仮想機械が提供する機能すべてまたはそれらの機能のあるものだけを提供したり，あるいは何一つ提供しなかったりすることができる．たとえば，第6章で述べた基本ファイル・システムは，入出力システムとハードウェアとを用いている．入出力システムとハードウェアとが提供するコマンドは，一つの仮想機械を定義する．明らかに，基本ファイル・システムのユーザは，その仮想機械が提供するようなコマンドを呼び出すことができる．基本ファイル・システムの下にある仮想機械，すなわち入出力システムにユーザも自動的にアクセスさせるかどうかという点は，設計上の選択の問題である．その選択権は基本ファイル・システムにまで残しておくことができる．すなわち，基本ファイル・システムは，自分自身のコマ

ンドに下に存在する仮想機械の提供しようと選択したコマンドを追加して提供する．別の方法として，仮想機械のユーザはその下に存在する仮想機械のすべてのコマンドを，自動的に使用できるようにシステムの構造を定義してもよい．この問題は，一般的な場合を考えることによって一番良く理解できるであろう．

$L(1), L(2), \cdots, L(n)$ を，ソフトウェアの層の系列としよう．ただし，$L(0)$ はハードウェア，L(1)はソフトウェアの第一層などとする．層 $L(i)$ は仮想機械 $L(i-1)$ が提供する機能にしかアクセスできないような場合を考えよう．$L(i-2)$ の機能が $L(i)$ で必要な場合は，$L(i-1)$ がその機能を提供しなければならない．もう一つの場合として，層 $L(i)$ は $L(1), L(2), \cdots, L(i-1)$ が提供するすべての機能にアクセスできる場合を考えてみよう．したがって，$L(i)$ が提供する仮想機械は $L(i)$ が実現するコマンドとともに，$L(i)$ が使用できるすべてのコマンドも含んでいる．これらの2つの場合の中間としては，$L(i)$ は $L(1), L(2), \cdots, L(i-1)$ が提供するコマンドのうちの一部分だけを使用できるようにすることができる．たとえば，あるシステムにおいては，ソフトウェアの第一の層がハードウェアの特権命令を発行することのできる唯一の層である．他の層は，非特権的命令と第一の層のコマンドを実行できる．

それぞれの場合，その利点と欠点とを持っている．一つの層が一つ下のレベルのコマンドをアクセスできるだけならば，各層の設計者はその前の層のことだけを気にかけていればよい．したがって，システムの重要な性質は自動的に分離される．内側の層がすべてアクセス可能な場合には，次のレベルの設計者は自分が基本操作として必要なコマンドをふるいにかけなければならない．設計の思想からみれば，前者の方が魅力的ではあるが，非常に効率が悪くなる可能性がある．たとえば，$L(1)$ が提供する機能を $L(i)$ で必要な場合には，$L(2), L(3), \cdots, L(i-1)$ はすべてその機能を提供しなければならない．これはその機能に対する $L(i)$ からの要求が，その要求を処理できる層 $L(1)$ に到達するまで，$(i-1)$ 個のソフトウェアを通過しなければならないことを意味する．この効率の問題により，第二の構造を採用する方針になるかもしれない．

中間の場合として，ソフトウェアの層を木構造とすることが考えられる．各層は木構造上の祖先に対応する層にだけアクセスできる（たとえば，図2-1参照）．実際に，ソフトウェアの木は仮想機械の木を作り出す．各仮想機械は，そ

の祖先によって与えられるコマンドをすべて持っている．第10章では，SUEシステムを参考にしながら，この構成をさらに議論しよう．

設計の上昇法においては，設計者は一時には一つのレベルに対して，その注意力を集中でき，現在のトップ・レベルの仮想機械から新しいトップ・レベルの仮想機械へ発展させることだけを設計すればよい．同時に，設計の決定は仮想機械の順序に従って行なわれる．第一に，ハードウェアに一番近い機構に関する決定が下される．次に，ハードウェアから進んだシステムの部分についての決定が下される．このようなレベルの様式に従って設計したシステムの例として，THE システムの概略を述べよう [Dijkstra, 1968b].

8.2.3 THE 多重プログラミング・システム

THE はページ化した記憶の下で，協力して働く順次的プロセスの集団から成り立つ多重プログラミング・システムである．一つの順次的プロセスは，各ユーザに対応し，また（バッファリングのため）入力と出力機器に対応する．“セグメント・コントローラ”プロセスには，ドラムが与えられて，主記憶とドラムの間でページを転送している．また．“メッセージ・インタープリータ”プロセスには，オペレータのキーボードが与えられている．システムは6レベルの仮想機械で構成されている．

レベル0では，停止していないプロセスに対し，処理装置を割り当てる．実時間の時計からの割り込みは，プロセスを時分割するためにこのレベルで処理する．この上のレベルでは，各プロセスは自分自身の仮想処理装置を持つことができる．すなわち，プロセスは，各々が自分自身の処理装置を持っているかのように独立に実行することができる．

セグメント・コントローラはシステムのレベル1にあり，コアとドラムの間のページ・スワッピングを処理する．このレベルより上では，概念的には一つのレベルの記憶があることになる．すなわち，ドラムとコア内のページとの差異は消えてしまう．

メッセージ・インタープリータは，レベル2にある．コンソール・キーボードはすべてのプロセスが共有する．プロセスがオペレータにメッセージを転送するときは，自分を識別しなければならない．同様にして，オペレータがメッ

セージを起動すると，そのメッセージを送るべきプロセスを識別しなければならない．メッセージ・インタープリータの責任は，各メッセージを適当なプロセスに送りつけることである．この上のレベルでは，各プロセスは自分専用のコンソール・キーボードがあるものと考えることができる．もしも，メッセージ・インタープリータをセグメント・コントローラの下に置いておいたならば，その下に存在する仮想機械はページングを支援してくれないので，それに対する永久的なコア記憶を確保しておく必要がある．

レベル3は，周辺機器の入力と出力の流れに対するバッファリングのプロセスから成り立つ．このレベルより上では，プロセスは論理装置と交信をする．入出力プロセスはオペレータと連絡をとらなければならない（たとえば，装置の誤動作を伝える）ので，メッセージ・インタープリータよりも高いレベルにいなければならない．

ユーザ・プロセスはレベル4であり，オペレータはレベル5である．

上昇設計法の主要な利点は，システムを構造化できるということである．異ったレベルの間には明確なインターフェイスが存在できるし，プロセスは低いレベルのプロセスからだけ処理を要求できるのであるから，予想もしないようなループは取り除かれる．上昇する順序づけは，システムの実現の場でも適用することができる．レベルもハードウェアから開始して，その順序に従って実現する．実現の各レベルは，構成しテストできるような新しい仮想機械を定義する．この方法を用いたテストのための手続きは，各レベルを独立にかつ完全にテストできるので，より徹底的なものとすることができる．一つのレベルを完全にテストすると，次のレベルの部分をつけ加え，テストを続行する．設計者は階層的に構成することにより，システムの正当性をずっと強く確信することができる．

疑問に思っていたかもしれないが，この設計法を用いたときの主要な難点はレベルの選択とその階層的な順序づけである．THEシステムにおいては，レベルは高度なレベルのプロセスへの制限された資源の割り当てによって，特徴づけられると思われる．プロセスの親子関係は，もし存在するならば，レベルを用いたシステムの設計構造に関係しているだろう（第2．3節参照）．SUE システムにおいては，親子プロセスの木は設計の過程で用いられている階層構造と

直接関係づけられている．

8.2.4 下降設計法

システムを設計するもう一つの方法は，ユーザの仮想機械に対して望むべき特徴から出発して，順次，精錬しながらハードウェアまで下りていくというやり方である［Parnas および Darringer, 1967］．設計者は，最初にユーザの仮想機械の仕様を決めることから始める．この仮想機械の実現は，完全には規定しない．未定の部分は，さらにシステムの構成要素として設計する．その作業はシステムをかなりよく精錬して，その基本的な機能がハードウェアとか ALGOL プログラムのように，ハードウェアによって直接実行できるモジュールによって実現できるまで続行する．

もう一度，橋の設計との類似を考えてみよう．橋はその機能の仕様をちょっと考えることによって，設計することができる．それはある距離を渡さなければならない．それはある量の自動車の交通を支えることができなければならない，等々である．次に，橋台の長さ，幅，強さ，塔の高さ，支える重量など，個々の部分の仕様をひき出す．そして橋台，構造，表面などの各部分を，その仕様に従って設計する．橋の設計は，必ずしも橋の建設と同じステップを踏むわけではない．実際の場面では，橋を建設するステップは物理的な法則によって決められる．設計の最中には，橋台に対して支えをつけずに図面を引いてもよい．実際の場面では，橋台は適当な支柱をつけずに，空中に浮いていることはできない．橋の建設を支配している物理的法則は，設計の中間のステップで，たとえば橋のモデルを作り，その特徴の全体的な設計をテストするようなことの妨げにはならない．

下降設計法は構成要素の入れ子集合を作り出す．設計の各点において，設計者は要素の機能的仕様の抽象化を行ない，その抽象化をさらに小さなもっと詳細な部分に精錬しなければならない．たとえば，ある人が下降法で計算機システムを設計しているとするならば，恐らくシステムの中で必要な機能を与える協力して働くプロセスの集合から開始するであろう．低レベルの抽象化においては，システムの手続きやデータ構造の仕様が指定されるであろう．各手続きやデータ構造は，その部分要素を設計することによって，さらに精錬される．

この設計は第8.2節で議論したように，本質的には抽象化の入れ子の系列になっている．

この方式を用いたとき，システムの構成要素をハードウェアかマイクロ・プログラムかソフトウェアにするかという決定は，設計の初期の段階で下すべきものではない．これは構成要素の処理の性質の実際の差異に基づくよりも，むしろ全体的な性能や使用の頻度に基づいて行なうべき問題である．

設計の展開を調べるためには，最初からシミュレーションを使うことができる [Zurcher および Randell, 1969]．この方式により，構成要素が**どのように**機能を実現するかというよりも，むしろ**どんな**機能を構成要素が実行すべきかということを，設計者が考えるようになる．システムの構成要素はその設計が進むにつれて，もっと詳細にモデル化することができる．構成要素の初期の表現は，その要素の動作をシミュレートするために，与えられた入力と時間遅れとに対し，適当な出力を作り出すアルゴリズムで与える．設計が進むにつれて，そのアルゴリズムは設計する要素の次の集合に対する呼び出しの系列の集合によって置き換える．発展の各段階においては，目的に合致しているかどうかを見るために，設計を評価する [Graham ほか，1971 ; Parnas および Darringer, 1967]．システムの各抽象化は，手続きの階層から構成されるシミュレーション・プログラムに対応する．このプログラムはこの抽象化のレベルにおいて，システムの状態に影響を与えるような変数を，どのように処理するのかを規定する．たとえば，与えられた抽象化に対応するプログラムPが，その上の抽象化のプログラムQによって制御されているものとする．Qは自分自身の変数に基づいて，全域的な決定を下すが，その変数は実際はPの変数の抽象化である．Qの変数における変更は，Pの変数における変更に対応する．同様に，Pの幾つかの手続きはもっと低いレベルの抽象化のプログラムに対して，その処理を実行することを要求する．

最終的に，システムが設計されたとき，一番低いレベルの抽象化における基本的アルゴリズムとシミュレーション・プログラムによって提供される機能とを，システムを作りだす機構と置き換えることにより，システムの実現が可能となる．これを適当に行なうと，システムは最高のレベルの抽象化において指定しているものと全く同じ機能を提供することができる．

シミュレーション・プログラムがシステムになるような，最終的なインターフェイスは大きな問題点となることがある．システムの各々の精錬は，きれいさ，簡潔さ，理解し易さなどを強調して行なう．低レベルの抽象化は実際のハードウェアとインターフェイスをとらなければならないが，そのハードウェアは扱いにくい．その結果，ハードウェアとの最終的なインターフェイスは，非常にすんなりとはゆかないかもしれない．

たとえば，主記憶の保護について考えよう．高レベルの抽象化においては，設計者はセグメントに応じて，記憶を可変長のブロックに分割し保護する．システムを IBM/360 ハードウェアにおいて実現しているならば，低レベルの抽象化においては，保護は2 *K*のブロックに対してのみ実現できるだけである．これは設計者が最初の高レベルの抽象化において無視することができない，実際の機械の構造上の詳細である．最終的なハードウェアとのインターフェイスにおける非能率さはシステムの全体の性能に対して，大きな影響を及ぼす．

下降法による設計を行なったとき，レベル分けによりシステムに対する変更の影響を局所化することができる．設計において変更をすれば，適当なレベルでの変数を調べる必要が出てくるが，最低のレベルでの変数は考慮しなくてもよい．ある抽象化のレベルで用いている一つの対象物の変更が必要なとき，その特別な対象物を支持している設計の部分だけを変更する必要がある．設計の残りの部分は同一のままにしておくことができる．たとえば，一つのモジュールが3つのプロセスA，B，Cによって構成されているとき，Bに対する仕様の変更はBの精錬に対してのみ影響を与える．AとCの精錬は変更しないでよい．高レベルの抽象化をシミュレートしているプログラムは，システムの健全な設計と性能とを検証するために実行することができる．このテストの環境においては，呼び出される多くの手続きは，何もしないですぐに戻ってしまう．その理由はそれらの手続きは未だ計算を実際に実行するコードを持っていないからである．このレベルのテストで，システムのいくつかの機構の論理的な正しさを検証することができる．これらの手続きを実際のコードで実現したとき，そのコードは仮定した仕様に直接従うように注意しなければならない．

別の方法として，システムの設計は下降法で，実現は上昇法で行なうことができる．全体の構造の設計は下降的に行なうが，モジュールは実際には上昇的

にまとめる．この方法は，上昇実現法の利点を持っており，下降設計法で得られる大域的見地も犠牲にしないで済む．

上昇設計法と下降設計法との提案者の間には，多くの議論が戦わされている．大きなまた重要な決定は早めにし，詳細な実現上の決定は遅めにするという設計哲学によりよく合致するのは，下降設計法であると多くの人は信じている．しかしながら，下降設計法は設計の初期の過程において，システムをよく理解することが必要になる．上昇設計法はソフトウェアの各層をすでに完全に仕様の決まっている仮想機械に対してつけ加えてゆくので，その理解は少くてもよい．その結果，特にオペレーティング・システムの場合，下降法を説くけれども実際には上昇設計法を用いる人々がいる．議論の焦点はどちらかの方法に対して有利な結論を出すことではない．この節，そして本章全体で強調したいのは，設計は完全にその場限りの方法である必要はないということである．設計の方式を技術的な利点に基づいて比較するという事実は，喜ばしいことである．ソフトウェアの設計は，芸術よりも科学の領域により近いかもしれないことを，それは暗示しているのである．

8.2.5 構造的設計法のおわりに

設計の決定を構造的に行なう方法は，他にもいくつか存在する．設計は，実在するハードウェアとユーザの仮想機械との間のギャップを橋渡しする仮想機械の階層と直接関係づける必要はない．一つの良い例は，“易しい決定は最初に行なえ”ということである．すなわち，システムの性能に大きな影響を与えないような決定は，すべて素早く下すべきである．もちろん，大した重要さを持っていない決定を識別することは，常に容易であるとは限らない．重要な決定を下したときに全体的な性能上の誤りが，偶然あらわれてくるかもしれない．反対のやり方としては，非常に信頼の少ないシステムの部分に集中するというやり方である．最初に難しいモジュールをこなしてしまえば，システムをよく理解できるが，それは形式的な上昇または下降法において，有益なものとなるであろう．システムのソフトウェアのある層が，非常に重要なので，それを最初に完全に設計してしまう時もある．そのような層の一つが，通常オペレーティング・システムの**核**（nucleus）と呼んでいるものであるが，それは基本的な

資源割り当て機能をすべて提供する仮想機械である [Brinch Hansen, 1970]. 核の主要な目的はプロセスとその通信機構とを実現し，ハードウェアを共有するための基本的機能をいくつか提供することである．たとえば，SUEシステム（第10章参照）は**核心**（kernel） と呼ぶ小さな部分が存在するが，それはプロセス，主記憶，保護の資格を扱う基本命令を提供している．SUEの核は入出力，基本ファイル・システム，資源の管理者を提供する仮想機能を実現する．核心と核とはオペレーティング・システムの設計の残りの部分の指標となり，制限条件を与えてくれる，固定した中間的な仮想機械を提供している．

きれいな設計方法は実用のためには使わないことが多い．時には，高いレベルの仮想機械が提供する操作を低いレベルで用いなければならず，したがってあきらかにシステムの階層構造に矛盾が生じてくる．たとえば，仮想機械のレベルに従って設計したシステムにおいては，すべてのハードウェアの誤動作処理のルーチンを主記憶に格納しておくのは，そのために非常に多くの空間が必要なのでむしろ非現実的である．それらのルーチンを周辺装置に格納しておけば，誤動作が生じたときに適当なルーチンを主記憶に転送してこなければならない．ファイル・システムはその動作を行なうために用いる．しかし，誤動作は通常入出力システムで検知するが，それはファイル・システムよりも低いレベルの仮想機械である．それゆえ，入出力システムはそれよりも上の仮想機械，すなわちファイル・システムによる処理を要求しなければならない．

構造化設計法は，適用することが時には困難であるからといって，それが不適当なものであるという意味ではない．構造的設計は設計過程を規定する道具である．しかし，ある特別の場合には，設計者は自分自身の充分な理由があり，自分自身が責任をとるつもりで構造を用いた技法から少し離れてみようと決心してもよいのである．

8.3 設計のすすめ方

前述した方法とか他のいくつかの一般的な設計方針に基づいて，オペレーティング・システムを構成するソフトウェアの設計を概説してみよう．基本的な仮定として，システムの機能の記述とハードウェア自体またはハードウェアの完全な仕様とが与えられるものとする．設計は5つのステップによって進行す

る．異ったステップを区別する概念的に重要なことは何もない．それらは設計の過程において，新しい方針を追加するために分離しているのである．このステップは順序に従って実行するけれども，いくらかの後戻りすることが頻繁に必要となる．設計の過程では，決定はある程度の不明確さを持って下す．それらは常に正しいわけではない．それゆえ，設計者は一つのステップを何回も繰り返さなければならないかもしれない：

1. **骨組の設計** (Skeletal Design)　システムの全体の骨組の設計は，異ったモジュールとその間のインターフェイスの第一近似として概略を述べなければならない．システムの機能的な仕様とハードウェアの機能とのギャップは，構造化したモジュールの集合によって橋渡しを行なう．第8.2.4節に概説したように，設計の過程は下降的に進行する．システムの核のような幾つかの仮想機械はあらかじめ定義してあってもよい．システムの中間的な抽象化は，シミュレーションかまたは少なくともその記述を非常に注意深く読むことによりその正当性をチェックする．
2. **モジュールの仕様決定とインターフェイス**　前のステップで記述したモジュールに対し，その正確な機能とインターフェイスとの仕様を決める．各モジュールは，最初は高いレベルの抽象化で記述してある．この記述を下降法的に精錬して，さらに各モジュールの仕様を決める．詳細な抽象化をだんだんに用いて，すべてのモジュールとインターフェイスとを記述する．この記述によりシステムの構成要素の振舞いを細かく調べることができる．デッドロックの可能性のようなシステム内のいくつかの論理的な問題点は，このレベルで検知することができる．
3. **一般的な機構の設計**　この段階では，次のような基本的な機構の詳細な仕様を決めるためにシステムの一般的な特性が充分わかっている．(a) プロセスの実現．(b) プロセスの通信．(c) 保護．(d) 記憶の割り当て．

   前のステップに対して，ある程度の後戻りが必要かもしれない．異ったモジュール間の相互作用にある条件をつけることにより特定の機構を改良することができる．たとえば，システム内のプロセスに階層構造を持たせることによりプロセスの通信機構を簡略化できる．

4. **モジュールの全体的な設計**　システム内の各モジュールは，この段階では独立に設計することができる．モジュールは，相互作用が明確に決められているべきいくつかのプロセスや手続きに分割できる．モジュール内の制御の流れと授受するパラメータとは，各モジュールの最初で明確に言明しなければならない．

ここまで来れば，システム全体の性能を考えることができるような詳細なシステムの見取り図ができる．システムの効率に大きな影響を与えるような頻繁に行なうべきある特定の操作に必要な時間について，何か簡単な計算をすべきである．特にシステムの性能が最初の仕様に合致しなければ，前のステップを繰り返すことが必要になるかもしれない．

5. **各モジュールの詳細な設計**　各モジュールは，独立に詳細に設計し，プログラムし，テストする．詳細な設計とプログラミングそのものとを区別するのは困難である．第9章においては，下降設計法の自然な拡張である下降的プログラミングの方法について述べる．

システムにおけるモジュールの統合は，注意深くテストするために上昇法で進めることができる．不注意な設計をすれば，テストの場面において問題が出てくるかもしれない．チームはしばしば並列的に作業しているけれども，モジュールはある特定の順序に従って統合しなければならない．その一つのモジュールがまだ準備されていなければ，他の個々のモジュールを統合したりテストしたりするためにシミュレートした環境が必要となってくる．

言うまでもないことではあるが，この最後の場面は微妙なところである．前のステップで正しく処理してなかったすべての問題が，いまや姿をあらわしてくる．概略を述べてきた設計のステップは何回も繰り返され，時には設計者が最初の仕様に戻ることを強制されることがあるかもしれない．これらの問題点は予測をつけておくべきである．実際には，上昇法にしろ下降法にしろ完全に事前に決められた順序に従って設計のステップを進めることは，めったにない．しかし，前に行なった設計が直接関係しないような，制御されていないステップの繰り返しは避けなければならない．新しい要求に対応するために連続的に変更をしてゆくと，そのような制御されていない繰り返しを招くことになる．

たとえば，営業の人々はシステムの仕様の変更を続けるかもしれない．その結果は，設計者やプログラマに完全な混乱を与え，プロジェクトの重大な遅延を招くことになる．

設計における重大な危険性の一つは，設計し過ぎる（over design）ことである．最終的な実現に影響を与えないように見えれば，余分な特徴を含めようとする誘惑が生ずる．余分な特徴はそれが必要になるまで排除しておくことが賢明な進め方である．使いみちのない一般性を取り入れることは，製品の性能とその実現のための複雑さの両方に大きな影響を与えてしまう．システムは拡張可能なように設計することが望ましい．そうすれば一般化や追加すべき特徴を，後になって組み込むことができる．

営業上の理由から，守ることが不可能な厳しくて非現実的な納期を課せられることがある．その結果，多くのその場限りの決定を設計上でしてしまう．最終的な製品は“めちゃくちゃな”設計によって生じた一貫性のなさからのがれることは出来ないであろう．システムのデモストレーション用の原型を部分修正することはできても，後で設計をすべてやり直すことが必要になってくるであろう．特に初期の段階では注意深く設計することに勝るものは何もないのである．どんなに部分修正の量を多くしても，悪く設計したシステムを改良することは不可能である．

設計過程の速度を上げるために，大きなオペレーティング・システムでは，従来多人数の手を借りてそれを実現してきた．適切に訓練されていない人達に，プロジェクトの仕事をたくさん割り当てている．人々の間の連絡および設計しようとしているモジュール間の連絡が一つの問題になる．実務の能力の大部分は，プロジェクトの管理に費してしまう．これらの見解から自然にプロジェクト管理の問題が生じてくる．

8.4 プロジェクト管理

ソフトウェアは，通常，最初に決めた時間と費用の範囲内では製作されないと言っても，恐らく控え目な言い方になるであろう．経験から考えると，多くの遅れと，費用の増加とがかなり頻繁に見られるので，それがソフトウェア製作の実際の姿だと思うことが多い．悪い管理やプロジェクトに影響を及ぼす不

確定な事件のつけを払うために，時間の推定に大きな安全係数をかけることをよく行なう．しかしこのやり方はよい工学とは考えられない．

ソフトウェアのプロジェクトを管理するのが困難であることの理由は，2つの考え方に従って分類することができる：

1. **貧弱な管理**　管理学と計算機科学の教育の間のギャップにより，適切な技術と管理の能力を持った人は非常に少ない．その結果，ソフトウェアのプロジェクトは，技術的能力はあるが管理の能力や経験が限られた人達か，または管理の経験は深いが実際には設計したシステムを理解できずまたソフトウェア・エンジニアとはうまく対話できない人達が管理することになる．
2. **定常的でない作業**　ソフトウェアの生産の計画を適切にたてることは非常に困難である．製作の仕様を変更することが多い．人事の変更は頻繁にあり痛手をこうむる．計画のための道具やデータは欠除している．ソフトウェアのプロジェクトを管理するためには，プロジェクトの性質を考慮に入れた特別な技法を用いなければならない．たとえば最終的な製品はもともと提案されたものよりも全く異ってしまうだろうということを，最初から認めていなければならない．

ソフトウェア設計のプロジェクトはどんな管理者にも悪夢を与えるような環境の中で行なわれることが時にある．たとえば橋の建設のエンジニアが一つの橋を設計するのに成功したときは，続けてもう一つの橋を設計することが期待される．ソフトウェアのエンジニアが一つのソフトウェア・システムを設計したときには，その人は通常別のもっと複雑なシステムに移るか，管理者の地位に移るか，あるいはアカデミックな環境に移るであろう．したがって，大部分のプロジェクト要員はプロジェクトの活動に対して新しい人々であり仕事をしながら教育も受けていることになる．

プロジェクト管理の全体的な目標は，言明した設計の目的と仕様内でかつ使用可能な資源以内で，望んでいる製品を作り出すことでなければならない．初期の問題は，会話のために自然言語の一つの共通な方言を見つけ出すことである．残念ながらソフトウェアの分野の術語は，すべての人々にとって同一の意味を持っているわけではない．実在するシステムの簡単に読める（！）マニュ

アルを，術語の最初の辞書として採用することは役に立つかもしれない．プロジェクトの要員は，プロジェクトの間話したり書いたりする言語の基礎として，そのマニュアルを用いてもよい．

適切なプロジェクトのドキュメンテーションは重要なことである．プロジェクトに対する作業帳を作ることは良いアイデアである．各プロジェクト要員はそのコピーを一つ持つべきであろうが，それは定期的に更新しなければならない．主要な決定，作業用論文，技術的論文，会議の議事録——プロジェクトにとって重要だと判明されるすべてのもの——を，作業帳は含んでいなければならない（要員の一人が，この更新の責任を持つことができる）．内容は周期的に濃くすることができる．古い版はうしろに設けた特別な節に入れることもできる．作業帳はプロジェクトの現在の状態を知らせるだけでなく，その完全な歴史も知らせてくれる．それゆえ，作業帳は適切なドキュメンテーションの基礎を形成している．それはまた，その完成あるいは失敗によって停止した後で，プロジェクトの反省をする時に役に立つであろう．

プロジェクトのチェックポイントは注意深く守らなければならず，またチェックポイントの報告書もつけなければならない．チェックポイントの報告書は，プロジェクトの重荷と考えてはならない．すべてのチェックポイントの報告書は，最初の目的，仕様，提案したプロジェクトの利益まで常にさかのぼって書くべきである．提案書を修正する必要が生じてきたら，設計作業を続行する前に書き直さなければならない．

チームが**小さい**（５～７人で昼食のテーブルに着くことができる位）ときは，全員により週に２～３回の会議を持つことが望ましい．一つの会議は技術的なもの，もう一つは非常に非公式的なものにすべきである．非公式的に頻繁に会話することも奨めなければならない． Parnas は要員の間で連絡をやり過ぎると，**モジュラリティ** (modularity) に悪い影響を与えると述べている [Parnas, 1971]． それは人々は非公式の情報を用いて， 標準的なインターフェイスを使用しなくなる傾向にあるからである．この問題に打ち勝つことは管理と方針の仕事である．

プロジェクト・チームが小さくないときは，非公式的な連絡はほとんど期待できない．そのプロジェクトに関するいくつかの疑問，たとえば何故そのプロ

ジェクトはそんなに大きいのかということを尋ねなくてはならない．それは，もっと小さくできないのか？　通常の連絡機構（たとえばメモなど）を使用しなくてはならない．しかし，プロジェクトの多くの能力を連絡のために使用しても良いということを，よく理解しなければならない．

実在するいくつかのシステムは，“大きな”チームの人々が設計してきた．“大きな”ソフトウェアのプロジェクトは，通常，25人以上加わったもの，あるいは少なくとも2つのレベルの管理のあるものと定義する．大規模なプロジェクトを管理する問題は，**ソフトウェア工学**（Software Engineering）に関するNATOの報告書の中で概略が述べられている［Naur および Randell, 1969；Buxton および Randell, 1970］．大規模なソフトウェアのプロジェクトに対する必要性は，だんだんなくなってくる傾向にある．未経験の人々を多数雇うかわりに，高度に訓練した人2～3人にプロジェクトを生産するよい道具を与えた方がよい．多くの人々がプロジェクトに加われば，生産性や総合的な知的出力は増加するように見えるけれども，生産性は逐には一定値に達する．利益が減少してくる点は，プロジェクトの管理の能力に応じて変化する．大規模プロジェクトで生産性が下る理由の一つは，5～7人以上の人が関係すると形式的な管理の構造が必要になってくることである．大規模なソフトウェアのプロジェクトは“能力があり柔軟で，勘容で，物知りで，非常に指導的で残念ながら稀にしかいない”ような管理者を必要とする［Naur および Randell, 1969］．そのような人々は実在するかもしれないが，ソフトウェアのプロジェクトの管理のためにそのような人々が使える状態にあると考えるのは幻想である．

“Conway の法則”に従えば，システムの構造はそれを作成した組織の構造に似ていると言われている［Conway, 1968］．システム内の異った機能のモジュールの間のインターフェイスは，そのモジュールを作業している集団の間のインターフェイスと同じ程度に良くも悪くもなるであろう．チームが小さければ適切な連絡があり，プロジェクトの要員は一つ以上の機能モジュールの設計に参加する．この様な状況は直ぐに役に立つことがある．全体の設計を理解している少なくとも1人の設計責任者（できれば2人）が居る可能性がある．この人はプロジェクト管理者であってはならない．どんな管理者もプロジェクトを管理し，かつすべての設計の詳細を監督下に置くことはできないからである．

システムの設計責任者は時間のかかるつまらない仕事からは解放すべきである．そうすれば，彼は他の設計者の考えに対する共鳴板になるべく自分の時間の大部分を捧げることができる．小さいプロジェクトでは要員は一つ以上のモジュールの設計に参加するので，システムを全体的に見ることから始まる．彼等は明らかに失敗するような局所的な最適化をすることはしなくなる．さらに形式的な管理の組織の必要性も少なくなる．充分少ない数の中で民主主義が働くことができる．

プロジェクトを小規模に保つためには，各設計者やプログラマの生産性を上げるように適切な環境を作り出さなければならない．設計者がその貴重な時間を設計のために費すことができるような一つの組織の作り方は，多くの注目をあびてきた．これは“**主任プログラマ制度**（chief programmer method）”である［Baker, 1972；Mills, 1971］．この主なる考え方は経験の深い能力のある人を，管理の仕事から実際の設計やプログラミングに戻してくることである．主任プログラマはその**支援プログラマ**（backup programmer）と共に大部分のプログラムを設計し書く．**プログラム・ライブラリアン**（program librarian）は，実行したりマニュアルを更新したりするようなすべての日常的作業を行なうことによりその上の人々を支援する．さらにチームは少数のプログラマを雇い，主任プログラマの仕様やインターフェイスに従って小さい個々のプログラムを開発する．最終的には，管理者とシステム・アナリストがチームを補充する．

主任プログラマ制度は，大部分のプログラムの設計を行なうような非常に能力のあるプログラマの周りに，小規模でよく組織されたチームを作ることである．プログラム自体は構造化した下降法で設計すべきである．チームの各構成員は，はっきりと定義した機能を持っている．ライブラリアンは実行や更新の情報をすべて扱っている．したがって，プログラムはもはや各個人のプログラマの所有物ではない．

主任プログラマ制度は，構造的プログラミングの技法と合わせてNew York Timesの情報システム・プロジェクトなどに実際に使用されて驚くべき成功を収めてきた［Baker, 1972］．チームの生産性は平均値よりもずっと上であった．プログラムはすぐに正しく働き出し，非常に費用のかかるテストの段階を除くことができた．組織化，計画，注意深い設計のために費してきた時間は，システ

ムの正しさを得るための最終的な段階でつぐなわれたのである．

8.5 おわりに

ソフトウェアのプロジェクトにおいて生じてくる組織上の問題を，いくつか論じてきた．オペレーティング・システムは，注意深い設計とよい組織とから生れた製品である．いくつかの進め方や設計法について概略を述べてきた．それらはきちんとした設計方法論を構成しているわけではない．逆に，精密な設計方法論を想像することはほとんどできない．設計は絶対的に管理したり制約したりすることができないような創造的過程なのである．それにはアルゴリズムがあるわけではないが，また全然制御しないことも駄目である．制約のついていない設計は制約をつけ過ぎた設計と同様に困難なものとなるであろう．その中間のどこかに，良い方法があるはずである．構造を利用すれば，その中間に到達するのに役に立つ．設計者は，充分柔軟性を持ってはいるが，システムの全体的構造によって導かれてゆくのである．

オペレーティング・システムまたは他のどんなソフトウェア・システムの開発においても，最も貴重な資源の一つは設計者の時間である．ソフトウェアの設計の道具は，設計者の能率を最大にするという意味で，非常に大切なものになる．しかし，いくつかの設計プロジェクトにおいては，ソフトウェアの道具は使用できないかまたはほとんど使用されない．それゆえ設計者は自分自身の能力にもっと頼らなければならない．残念なことに，才能とか設計能力は乏しい資源である．多くの機関では，設計から不必要な負担を取り除くような設計の道具を作り出すことを追及する研究活動を行なっている．PEARL プロジェクト [Snowdon, 1972]，LOGOS プロジェクト[Rose, 1972]，ISDOS プロジェクト [Teichrow および Sayani, 1971] などでは設計しているシステムの記述，検証，解析のための道具を，設計者に提供するような環境を作り出すことを目指している．

演習問題

8.1 ソフトウェア・システムにおける2つのモジュール間の結合度の基準を定義せよ．

8.2 読者の機械のオペレーティング・システムを記述するような仮想機械の系列を述べよ．それらの仮想機械は，どのように関連し合っているか．FORTRAN コンパイラのような与えられた言語プロセサに付随している仮想機械は，どのように記述するか．

8.3 ハードウェアの設計者は自分自身の設計方法論を持っている．彼等の方法を調査せよ．それらは構造的にできているか．それらはソフトウェアの設計に似ているか．

8.4 自動的なインターフェイス・システムの文献が発表されている [Weissman および Stacey, 1973]．システム内のモジュール間の標準インターフェイスを提供するために，第2章で述べたようなモニタの使用の概略を述べよ．例を示せ．

8.5 多くの工学的なやり方では，設計者に非常に役に立つ過去の経験から得た多くの所見を集めたハンドブックを用いる．ソフトウェア工学のハンドブックに対しては，どんな内容が適していると考えるか．

8.6 仮想機械の階層として構成されているシステムを考えよう．いくつかの高レベルの仮想機械から隠しておきたいような，低レベルの仮想機械の持っている機能の例をいくつかあげよ．どうやってそれを実施するか．わく組として，ファイル・システムの異ったレベルを用いることができる．

8.7 8.2.3節に述べたTHE システムのレベルを別の方法で順序づけることができるか．答の正当性を示せ．

8.8 プロジェクトの PERT による分析は，多くの分野における管理の主要な道具である．しかし，この設計用の道具はソフトウェア・プロジェクトにおいては，あまりたくさん使用してこなかった．何故か．

8.9 2.3節で概略を述べたプロセスの階層と仮想機械の階層との関連性を論ぜよ．プロセス間の相互通信の役割は何であるか．第10章で述べるSUEシステムを考えよ．プロセスとその相互通信は，どのようにしてシステムの仮想機械とその構造とに関係づけられているか．

8.10 GPSS, Simula 67, Simscript のようなシミュレーション言語の性質を調べよ．設計の途中でシステムを評価するのに適した言語を，一つ選択せよ．その選択の正当性を示せ．

8.11 2つのプロセスの相互通信の機構，たとえば郵便受けとファシリティを比較するような実験を，SUEシステムと関連づけて実行せよ．設計の段階におけるシミュレーションの使用に関して，その実験からどんな結論が引き出せるか．

第 9 章

実　　　　　現

9.1 はじめに

ソフトウェア・システムの開発においては，設計と実現との間には明確な区別はない．たとえば，下降法によりシステムを構造的に開発したとき，設計は具体的な実現にスムーズにかつ徐々に移行してゆく．唯一の差異は，恐らく抽象的または非形式的な記法を用いるかわりに，プログラミング言語を用いてシステムの実現を記述することである．しかしながら，適当なプログラミング言語を選択すれば，それを設計の段階でも用いてシステムの部分を書くことができる．設計と実現を主題にして別々の章で論ずるのは，最終製品が異った特徴や性質を持っているということを強調するためである．

ソフトウェア・システムあるいはそれに対する任意のプログラムに対して，望ましい性質とは何か．最低限それは信頼性があり，効率がよく，わかり易いものでなくてはならない．オペレーティング・システムは信頼性のあるものでなくてはならない．すなわち，信頼の置けないソフトウェアはそれがどんなに多くのサービスをしてくれるように装っていても，ほとんど価値がないのである．ある種の応用に対しては，失敗によってかかる費用の方が計算機システムの費用よりもずっと高いことがある．たとえば，信頼性のないソフトウェアはシステムやユーザのデータを警告なしに破壊してしまうような隠れた誤りを持っているかもしれない．失敗により，複製することが非常に高くつくデータに影響を与えるかもしれない．ソフトウェアは信頼性を持たせるために，注意深く設計し実現しなければならない．信頼性は，突然，生ずるわけではない．Randell が述べているように，信頼性は追加してできるような性質のものではない．は

じめから不安定で信頼性のないシステムを改良するのは非常に難しい．誤りを修正するために変更すると，不注意によって別の誤りがしばしば発生する．

オペレーティング・システムは効率よく動かねばならない．オペレーティング・システムは機械の時間の大きな部分を占め機械の資源を大量に消費するので，その効率は計算機システムの全体の効率を大きく左右する．しかしシステムの別々の部分が全体の効率に対して等しく影響を与えるわけではない．実際，オペレーティング・システムの非常に小さい部分が処理装置の時間の大部分を使用しているという報告がある．それゆえ，システムのある部分に対しては，効率の要求は他の部分よりもずっと重要度が高いのである．

便利さを追及すると，効率の重要性がしばしば無視される．技術的な工夫のおかげで機械の容量や速度に余裕が出てくれば，それが実行できる実際の作業の量を増加させるよりも，むしろその余裕を機械をもっと便利にすることに用いることがしばしばある．信頼性に対する考慮も，時には，効率よりも優先することがある．それでも応答の速度が重要であるような多くの応用，たとえば実時間システムとかプロセス制御システムのようなものが多数存在する．信頼性，効率，便利さに与える相対的な重要度は，システムが提供しなければならない処理の型によって決めるべきである．

オペレーティング・システムの理解し易さは，特に重要である．プログラムは機械が実行するために書くだけではなく，他の人々が読むためにも書くのである．オペレーティング・システムは連続的に発達してゆく．すなわち，新しい拡張を定常的にとり入れてゆく．さらに保守のためには，広範で詳細なドキュメンテーションが必要となる．プログラムがその読み易さをますような“良い”性質を持っているならば，システムのドキュメンテーションを作ることは簡単であろう．たとえばプログラムをうまく節に分けたり，適当なコメントをつけたり，記号による変数名を利用したりしなければならない．

信頼性があり効率がよく理解し易いオペレーティング・システムを作ることは容易なことではない．これらの特性を達成するためには，自覚した，また時には苦痛な努力をする必要がある．プログラマは実現に対する責任があるので，プログラムのための道具はある方針を強制しなければならない．あきらかにプログラマが用いる主要な道具は，プログラミング言語である．言語はプログラ

ミングの様式に影響を及ぼし，したがってプログラムの性質を左右する．それゆえプログラミング言語の選択は，実現を担当するグループが最初に決定を下すべきことの一つである．

9.2 実現のための言語の選択

伝統的には，オペレーティング・システムをアセンブリ言語で書いてきた．こうすれば高級言語で書くよりもずっと効率の良いプログラムを得ることができると考えていた．使用できるプログラミング言語のかなりの部分は，科学計算や事務計算に向いたものであった（たとえば FORTRAN や COBOL)．それらの性質はシステム・プログラミングにとって，特に魅力のあるものではなかった．最近になって，いくつかのオペレーティング・システムを高級言語で書くようになった（たとえば，PL/I で書いた MULTICS [Corbato, 1969]）．そして，システム設計者はアセンブリ言語に比べて多くの利点を見出した．これらの発見により，システムの実現に高級言語を用いる場合の相対的な利点を論ずるまでになってきた [Sammet, 1971]．

高級言語は，よりよいプログラムを書けるようにプログラマを助けることができるような構造を持っている．プログラマはよいアセンブラのコードを生成するための詳細な部分を心配する必要はないので，高級言語を用いればアルゴリズムを簡単にプログラムすることができる．原理的には，プログラマは一日に付き同一の数のデバッグしたアセンブリ言語か高級言語の文を書くことができるので，高級言語を用いれば実現の速度を上げることができる．しかしながら，主要な利点はプログラムを書いている時に得られるのではなく，むしろ理解するとき，テストするとき，デバッグするときに得られるのである．高級言語におけるプログラムはそれ自体でドキュメンテーションになるので，プログラマは相互にリスティングを読むことができる．これはプログラムを保守するときに特に重要である．

高級言語に反対する主要な論点は，実行の速度が遅くなるかもしれないという点である．しかしながら，システムの各部分の実行の頻度を解析すると，通常，ほんの２～３のモジュールがオペレーティング・システムの実行時間の大部分を占めていることがわかる．それゆえ，大部分のモジュールをアセンブリ

言語でコーディングするという特別な努力は無駄である．頻繁に使用するモジュールは，そのことが一度わかればもっと効率のよいものに書き直すことができる．MULTICS システムにおいては，ページ・フォールトの機構のようなシステムの少数の部分をアセンブリ言語で書き直すことによって，非常に大きな速度の改善が得られた［Saltzer および Gintell, 1970］．保守とかドキュメンテーションのために，このアセンブリ言語のコーディングのある部分は，速度から見て同様の改善が得られるように後ほど高級言語（PL/I）で書き直した．したがって，改善はアセンブリ言語を用いたからというよりも，むしろ微妙なコードに対し特別な努力を集中したことによって得られたと思われる．アセンブリ言語のプログラムは局所的な効率に過度な重点をおいて，全体的には効率が悪くなるという傾向にある．

実現のためには高級言語を用いるものと仮定して，問題はどれにするかである．主としてシステム・プログラムのために設計した言語，たとえば BCPL, PL/360, LIS, BLISS, LSD, SUE など面白そうなものがたくさんある．さらに，システム用言語の人口は，毎年，増加し続けている．いくつかの汎用高級言語たとえば PL/I とか PASCAL もそのよい候補となっている．一つの特定の言語を客観的に選択するのは難しいけれども，選択のための基本的な指標の概略を述べることができる［Atwood ほか, 1972］．

システム用言語はきれいなプログラミングの様式を実現できなければならない．制御構造が簡単に読み取れ，正当性を示すのに容易であるような，簡潔なプログラムを書くために便利でなければならない．データ構造は，柔軟で，強力で，直観的でなければならない．ＧＯ　ＴＯ文はその一般性と制御できない性質とを持っているので，多くの批判を浴びてきた［Dijkstra, 1968c］．それはプログラムの多くの完全に異った状況において使用することができるが，そのためにプログラムの読み易さを大幅に減少してしまうことがある．

システム用言語は読み易くなければならない．能力の大きいいくつかの言語で書いたプログラムは，それを書いた人以外には完全に理解できないものがある．言語によって不可解なプログラミングを奨励するようなことがあってはならない．むしろ，言語によって，構成要素による式の明確さ，命名法，コメント機能，句割りの方法などが奨励されなければならない．

システム用言語は構文上および論理上の誤りの検出や分離に，積極的に役に立たなくてはならない．言語の要素は誤解や誤りをしやすいものであってはならない．さらに，言語の中に取り入れた冗長性を利用して，コンパイル時あるいは実行時のチェックの機能が使用できなければならない．

システム用言語は効率の良いコードにコンパイル可能でなければならない．言語は悪いコードを本質的に生成するような，能力の大きい要素を取り入れることはできない．またプログラマが機械の中で“自然”だと思われるような要素を記述するのに困難さを感ずるほど制限を強くすることもできない．プログラマは機械の特性について，通常は気にかける必要はないけれども，システム用言語は必要な時には命令の発生と記憶とレジスタの割り当ての両方を精密に制御することができなければならない．

言語はそれによって書かれたプログラムに大きな影響を与える．プログラマは，通常別のプログラミング言語で同じような誤りを犯すようなことはしない．ある種の構成要素は非常に頻繁に誤りの原因になる．特徴的な誤りの理由は，要素の複雑さ，貧弱な定義，不自然な振舞いなどに関連している．前述の指標はよいシステム・プログラミングを作ることを妨害してしまうような言語の特徴を分離するのに役に立つ．

たとえば，Ichibiah と Rissen [1971] によって論じられている ALGOL Wにおける CASE 文を考えてみよう．CASE 文は**図 9-1** のような形をしているが，そこでは I=1, 2, 3 の値によって動作A，B，Cが実行される．CASE 文によって頻繁に生ずる誤りは，場合分けが抜けたり場合分けの再配置を行なうことによって生ずることが多い．

さて，**図 9-2** に示すようなCASE 要素の修正版を考えよう．ここでは，COLORというラベルの SPACE を (RED, ORANGE, GREEN) と定義している．

```
CASE I OF
  BEGIN
    A;
    B;
    C
  END
```

図 9-1　ALGOL W の CASE 文

```
SPACE COLOR : (RED, ORANGE, GREEN) ;
'TYPE COLOR LIGHT;
        :
CASE LIGHT/COLOR OF
    GREEN :    A;
    RED :      B;
    ORANGE :   C;
END
```

図 9-2 CASE 文の修正版

変数 LIGHT は，COLOR という型と定義しているが，それは値として RED, ORANGE, GREEN のみを取るという意味である．場合は，番号ではなくそのラベルによって選択する．したがって，それらは容易に再配置したり，一つ除去したりすることができる．CASE 文の修正版では，場合に付与した番号はプログラマには知らさない．コンパイラはプログラマには見えない番号を割り当てる．したがって，修正した CASE 文は場合の順序づけに関する問題を除去することができる．

言語の選択は言語の構成要素に加えて，使用できるコンパイラのことも考慮に入れなければならない．コンパイラは，特にその言語が適切な性質を持っていれば，チェックをしっかり行なうのに適した位置にある．少なくとも，コンパイラは句切り符号の欠落など事務的な誤りの大部分を捕えなくてはならない．

たとえば，**図 9-3** の FORTRAN プログラムを考えよう [Elspas, ほか, 1971]．プログラムは構文的には正しいけれども意味がない．Z の値は前の値を全然使用せずに変更されてしまう．このような矛盾は，コンパイル時に発見することができる．

```
Z = 算術式
A1
A2
 :
AN
Z = 別の算術式
```

図 9-3 プログラムの例．($A1, \cdots, AN$ は Z を参照せず，分岐命令も出さず，GO TO の対象にもなっていない)．

明きらかに，ある種の意味的情報は実行時までわからないので，コンパイラは完全な信頼性を保証することはできない．それでも，言語の構文における有益な冗長性はコンパイラがあやしげに見えるコードを捜し出すいとぐちとして役に立つ．言語に課せられたある種の制限により誤りの検出に際して役に立つ情報をコンパイラは得ることができる．しかしながら，言語の制限と冗長性は，時には，非常に柔軟で能力の大きい言語構成という哲学に矛盾することになる．

残念ながら，プログラムの振舞い，特に，共通の誤りに関することはほとんど知られていない．プログラマ自身は，何が言語に関して悪いのか常に判断できるわけではない．異ったプログラミング言語を用いたときに生ずる誤りの形や頻度について，実験的なデータが必要となる．この種の情報はよいプログラムをもたらすような言語の性質を選択する際には，価値がないかもしれない．今日まで行なわれている言語の評価は，大部分，様式と明確さとに関する主観的な議論である．言語の設計者は，言語の設計の際に議論の基礎とすべき言語の使用に関する事実をほとんど知らないのである．

プログラミング言語は，実現の際に最もたくさん使用する道具である．その性質は最終的な製品の性質を大きく左右する．しかしながら，他のどんな道具でも同じように，それを正しく使用しなければならない．この観点から，自然にプログラミングの様式という話題がでてくる．

9.3 プログラム工学

プログラムの質を判断する方法は，たくさん存在する．プログラムの性能は信頼性，プログラムの明瞭さ，出力の明瞭さ，記憶の使用率，実行時間，保守の容易さ，変更の容易さによって測ることができる．実現の場面においては，プログラミングに要した時間のようなプログラミングの努力の量は，プログラムの質の量と同じように最適化することができる．これらの目的のうちのいくつかは，たとえば，プログラムの読み易さと保守の容易性のように相入れるものである．他の目的は，たとえば数の少ない文と開発に要する時間のように相反するものである．目的の選択は，最終的な製品の質に大きな影響を与える．

同一のプログラムを異った目的をもって作っているいくつかのプログラミン

グ・チームの間では，その達成したものがかなり異っていたという実験がある［Weinberg, 1971］．たとえば，2つのプログラミング・チームP［速い［prompt)］とE［効率よい (efficient)］に，異った目的を持つ同一の課題を与えたとしよう．Pチームは動くプログラムを出来るだけ速く作らねばならず，Eチームは非常に効率の良いプログラム(すなわち，処理装置の時間の少ないもの)を作らねばならない．Pのグループは39回の実行を予測し，29回目の実行で終了した．Eのグループは22回の実行を予測し，69回目で終了した．両チームともに同等に質のよいプログラマがいた．この実験によれば，プログラミング・チームに与えた目的はその性能と時間の予測とに大きな影響を及ぼすことがわかる．プログラマの経験や性能とは無関係に，前に述べた見解が成り立つように見える．それゆえ，プログラミングの目的は非常にまじめに受け取めなければならない．残念ながら，ほとんどの場合，目的は，はっきりと述べられていず，矛盾が生じたとき相対的な荷重を指定することができない．したがって，プログラミング・チームは遅れや費用のかかる再実行を引きおこすような目的を任意に決定してしまうのである．もう一つの実験では，同一の問題を異った目的で行なう課題に対して，コアの量や文の数に関して5倍も異なる結果をプログラミング・チームが示した［Weinberg, 1971］．注意すべきことは，すべてのチームは，自分達に与えられた目的に関しては非常に高い位置にランクされていたことである．自分達に要求されたことは達成している．これは“良い”プログラマと“悪い”プログラマとの区別の問題ではなく，むしろ，望ましい目標を選択することが問題であることを示している．

プログラマ自身の性能は彼等の教育，経験，能力によって変化する．“良い”プログラマは“悪い”プログラマの30倍の生産性をあげることができるという定説が広くゆきわたっている．しかし，良いプログラマが何をなすべきかということを概説するのは非常に難しい．行なわれたプログラミングの量を考えると，良いプログラミングを行なうためには何をなすべきであるかということに関しては，驚くほど少ししか知られていない．プログラム工学の目的は，かなり能力のある人々が正しい道具と手続きとを使用して，彼等の能力を高めることができるようにすることである．

プログラム工学を成功させるための道具や原理を生み出すために，現在，多

くの実験が行なわれている．この分野における一つの主要な功績は**構造的プログラミング**の考え方［Dijkstra, 1968c；Dahl ほか, 1972；Mills, 1972］である．構造的プログラミングとは下降法を用いてよく組織化された方法でプログラムを開発することである．これをGO TOなしプログラミングと混同してはいけない．GO TO文なしの言語でも，構造的でないプログラムを記述することはできる．一方では，よく組織化されたプログラマは，たとえばマクロを適切に使用して，アセンブリ言語でよく構造化されたプログラムを作ることができる．

本書の目的のために，次のような定義を採用しておこう：

1. **GO TO なしプログラミング**とは，GO TO 文を使用しないでプログラミングを行なうことである．たとえば，GO TO を制御文としていない言語を用いてプログラミングを行なう．
2. **構造的プログラミング**とは，下降法でプログラムを開発する高度に組織化した方法である．プログラムの記述が次々と精錬されて，ついには記述の抽象化のレベルがプログラミング言語のレベルに達する．
3. **構造的プログラム**は，適切な制御文とか句分け規則，変数名などを使用した結果得られた，明快で理解し易いプログラムのことである．

構造的プログラムは，通常，適切な特性を持った言語において，構造的プログラミングを介して作り出す．しかし，GO TO 文の削除などの良い性質を持った言語でも，良いプログラムを保証してくれるわけではない．たとえば，次のようなFORTRANの抜すいを考えよう［Kernighan および Plaugher, 1973］

```
      DO 10 I=1,N
      DO 10 J=1,N
10    V(I,J)=(I/J)*(J/I)
```

このプログラムには GO TO 文がない．一方，これは不明瞭で，神秘的で，非能率でさえある．このプログラムの機能はすべての行列のエントリを0にし対角線のエントリを1にして単位行列を作ることである．このプログラムはFORTRAN の整数演算における打ち切りの性質に頼ってこの結果を達成して

いる．同様の結果はもっと不器用ではあるが，ずっと良いプログラムによっても得ることができる．

```
      DO 20 I=1,N
      DO 10 J=1,N
 10       V(I,J)=0.0
 20   V(I,I)=1.0
```

構造的プログラムでは，最初に高レベルの抽象化において，プログラム全体の機能を記述する．次に手続きやデータ構造を詳細化するように再定義してゆく．プログラムの記述の詳細さのレベルがプログラミング言語に達したときに，そのプログラムは終了する．次の例はこのやり方を示している［Mills, 1971］．プログラムの機能の仕様は**メンバをライブラリに追加せよ**ということである．機能に対する識別名は１文字として展開している．太字の句はプログラミング言語を用いて内容のある記述に展開してもよい．

仕様（レベル　０）

f ＝**メンバをライブラリに追加せよ**

f は次の様に展開する：　g THEN h

（すなわち，最初に g を実行し，次に h を実行せよ）

部分仕様（レベル　１）

g ＝**ライブラリのインデックスを更新せよ**

h ＝**メンバのテキストをライブラリのテキストに追加せよ**

g は次の様に展開する：　IF p THEN i ELSE j

（すなわち，もし p が真ならば i を実行し，そうでなければ j を実行せよ）

部分仕様（レベル　２）

p ＝**メンバ名が，インデックスに入っている**

i ＝**テキストのポインタを更新せよ**

j ＝**名前とテキストのポインタをインデックスに追加せよ**

２つのレベルの展開を言い直すと，

$f=$(IF　**メンバ名が，インデックスに入っている**
　　THEN　**テキストのポインタを更新せよ**
　　ELSE　**名前と，テキストのポインタを**
　　　　　インデックスに追加せよ)
　　THEN　**メンバのテキストを，ライブラリのテキストに追加せよ**

上の例のかっこに入っているような機能の抽象化は，プログラムの低レベルの記述を理解するときには非常に役に立つ．各レベルにおいて，機能をあらわす文字の仕様はプログラミング言語の文，手続き，手続きの集団，その機能は何を行なうのかという日本語での記述のどれかにすることができる．抽象化のレベルがもっと詳細になると，機能の記述はプログラミング言語にずっと似てくる．さらに各抽象化のレベルにおいて，全体のプログラムが定義されているので，プログラムはその開発の間何回でも手で確かめたり，シミュレーションすることができる．

構造的プログラムはかなり漠然とした概念であり，通常，構造的プログラミングの産物として得られる．しかし，ある程度まで構造的プログラムの構造は心理学的な問題である．構造は見ている人の目の中にある．すなわち，常に，検知できるわけではない．構造的プログラミングは高度に組織化した方法でプログラムを作り出す．結果として得られたプログラムの構造はマクロ機能を介して覚えておかなければ，偶然読んだ人にとってはすぐにわかることもあるだろうし，わからないこともあるであろう．記号名称，制御構造を強調する区分化の厳密な規則，不明瞭なコードを説明するコメントなどの武器はプログラムの構造を表面に押し出し，そこで観察できるようにするためにすべて役に立つ．

現在のところ，プログラムの読み易さと理解し易さとを非常に強調している [Weinberg, 1971]．プログラムを読むことは，非常に面白い訓練である．Weinberg は，プログラマはプログラミング言語の中でプログラムの読み方を学ぶ前に，プログラミング言語の書き方を学ぶという逆説的現象が見られると報告している [Weinberg, 1971]．プログラマはプログラム（または日本語）を読めないのに，どうして立派なプログラム（または詩）を書くことが期待できようか．構造的プログラムは構造的でないものに比べて，一般に，読み易い．

McIlroy が指摘しているように，よく読めるプログラムは恐らくうまく走るであろう．これは原因と結果の関係を述べているのではない．むしろ，簡単に読むことのできないプログラムは，たいていの場合，プログラマが気が付かない隠された誤りを持っていたという見解を述べているのである．そのような誤りの発生はプログラムの効果が簡単に理解できるときは，ずっと頻度が少なくなる．

構造的プログラミングの技法は，データ構造の設計においても同様にうまく適用することができる [Dahl ほか, 1972]．データ構造はオペレーティング・システムにおいては非常に重要である．実際，オペレーティング・システムを設計し実現するとき，データ構造とプログラムとを同時に設計しなければならない．多くのプロセスと手続きとが同一のデータ構造を共有するような場合には，プログラムの設計の前にデータ構造の設計を先に行なった方が良いことがある．プログラムにおけるデータの構造が適当かどうか調べる一つの方法は，制限のついていないポインタを使っていないかどうかということである．制限のついていないポインタは GO TO 文に似ている．すなわち，プログラムを読んでもそのポインタを用いた結果生ずる全体的な効果を予測するのが困難である．データ構造を明確に定義し，スタックに対する “POP” とか “PUSH” のように，そのデータを検索するために用意した操作を使用する方がよい．

いくつかのシステム・プログラム用言語の設計者はデータ構造の重要性を認識している．たとえば，PASCAL においてプログラマが定義するデータ型のようにデータを構造化する道具をたくさん開発してきた [Wirth, 1971]．それらの道具は，プログラマが自分のデータ構造を処理する高級な操作を定義することを可能にしている．たとえば，いくつかの言語ではプログラマはデータ構造を自由に定義し，それらの基本操作に対する手続きを用いて実現することができる．

構造的プログラミングを用いると，最初にプログラムを書くための速度は低下するかもしれないが，テストや検証の場面で必要な時間が大幅に減少する [Baker, 1972]．プログラマの生産性が向上し，彼等が作るプログラムはほとんど一様に良いものになる．構造的プログラミングを用いると，プログラマは問題を全体の立場から見て，異ったモジュール間の関係を深く理解することができる．結果として得られるプログラムは理解し易く記述し易い．その正当性は

ずっと容易に検証できる．プログラムの設計過程において，プログラムに重ね合わせた構造の記述はプログラムの振舞いに対する良いドキュメンテーションになっている．これによりドキュメンテーションと保守との努力を減少できる．最近の結果によれば，この方法は実用的なものになってきた［Baker, 1972］．構造的プログラミングは有効なプログラミング技法へ向かう第一のステップとなるきざしがはっきりと見えている．

良い品質のプログラムは，簡単には生まれてこない．悪いプログラムを書く方法はたくさんあるが，良いものを書く方法は非常に少ない．良いプログラムを書くことは，悪いプログラムを書くのに比べて頭脳に負担がかかる．この事実を知ることにより，うまく行なうためには，まじめに考えなくてはならない様な重要な活動に従事していることに，プログラマは気がつかなければならない．

9.4　プログラムの検証

プログラムの検証はいくつかの局面から成り立つ：

1. **推測による分析**とは，プログラムを機械で実行する前に，注意深くプログラムの動作を吟味し，分析することである．
2. **テストすること**は，プログラムにデータの標本を入力して実行を開始し，誤りを見つけ出す作業のことである．
3. **デバッギング**とは，テストの際に見つかった誤りの原因を見つけ出し，それらを修正するためにプログラムを変更する作業のことである．
4. **確認**とは，通常，プログラムを書いた人によって，最終的なチェックを行なうことである．
5. **検査**とは，製品として使用される前に，プログラムを書いた組織がそれをチェックすることである．

この5つの局面はすべて重要である．しかし，プログラムの検証のための主要な道具は，テストすることおよびプログラムの正当性を証明することである．

テストをあますところなく行なえば，プログラムの正当性を検証できるかもしれない．しかし，一般にはほとんど実用的ではない．Dijkstra が指摘しているように，テストすることにより誤りが存在することは示せるが，それが無い

ことを示すことはできない．ある程度有効になるためには，テストを高度に組織化された方法で実行しなければならない．一つの実用的なやり方は，プログラムの制御の流れをすべてつくすことを集中的に行なうことである．コードの一部が実行されて誤りが存在すれば，最終的な結果（あるいは，プリントされるならば，中間結果）は影響を受けるはずであるという，本質的な仮定をしている．あきらかにこの仮定が正しくないような場合も存在する．しかし，通常，制御の流れをすべて覆うことは誤りを捕える一つのよいきっかけである．それによってプログラムの正当性は証明されないが，しかし，そのために明白な誤りを多数除去することができる．もう一つの簡単で有効な手続きは，例外条件たとえば変数の値域の極値をテストすることである．経験によれば，極端な条件（たとえば，待ち行列が空の場合）はプログラムを書くときしばしば無視されることがある．

必要な努力をもっと少なくするやり方は，各モジュールを独立にテストすることである．次の論点から，このやり方をすすめることができる．一つの制御の流れをテストするのに，一単位の作業が必要だと仮定しよう．n個の流れを覆わなければならない手続きを考えよう．別の手続きのm個の場所から，それが呼ばれると仮定しよう．すなわち，呼び出しを含む流れがm個あるものとする．これらの手続きを独立にテストするならば，最初の手続きの流れをテストするためにn単位の作業がまた第二の手続きのテストに対しm単位の作業が必要になり，全体で $n+m$ 単位の作業が必要になる．一方，両者を一緒にして考えると，結合したプログラムには$n \times m$ 個の流れが存在し，$n \times m$ 単位の作業が必要となる．あきらかに，テストを独立に実行すれば，作業は少なくて済む．

モジュールを独立にテストすると，モジュールを結合するまではインターフェイスにおける問題点はあらわれてこないかもしれない．たとえば，前述の例で呼び出される手続きは，呼び出す側の手続きに対し副次的効果をもたらすかもしれない。結合したプログラムの全域的なテストをすれば，この種の問題は発見される．しかしながら，すべての可能な流れを覆う全体のプログラムに対するテストの場合を見つけることは，特に望ましくない副次的効果を持っている場合は，困難であろう．システムを仮想機械の系列として構造化してあれば，各レベルのモジュールは高いレベルのものとは独立に検証することができる．

これらのモジュールは，さらにテストをしないで次のレベルの検証のために使うことができる．インターフェイスをあらかじめ厳密に定義してありシステム内で一様なものであれば，モジュールに関するインターフェイスの問題はそのモジュールより上のレベルでは生じてこないと保証してもよいであろう．

9.4.1 プログラムの正当性

アルゴリズムをあらわすプログラムは，数学的に正しいものであると証明することができる．この話題に関する研究は，本質的に2つの方法をとる．第一の方式は，プログラムが意図していることを行なっていることを示すために伝統的な数学的証明を手で行なうことである．第二のやり方は，数学的モデルを用いてプログラムの性質を形式的に述べ，述語論理の式を用いてプログラムの正当性を表現し，**数学的定理証明システム** (mathematical theorem prover) [King, 1969] によって自動的に証明する方法である．後者の方法は，概念的には非常に面白い。残念ながら，定理証明システムの現状からみれば，システムの正当性の証明に使用するのには適していない．

手で行なう証明は von Neumann の時代にまでさかのぼる．しかし，その実用上の重要性はほんの最近になって認識されたところである [Floyd, 1971 ; Hoare, 1971 ; London, 1970]．原理的には，それは全く単純である．プログラムのいくつかの場所，たとえば $p(1), \cdots, p(n)$ において，プログラマはその時点において変数間で成り立つ不変の条件の**命題** (assertion) を与える．プログラムの入力変数に対する命題が少なくとも一つ，可能な制御の流れに対して少なくとも一つ，プログラムが作り出すものを規定するような出力変数に関するものが少なくとも一つ，なければならない．**検証過程** (verification process) は次のようなステップをふむ．

$p(i)$ が命題で，$p(j)$ はある制御の流れにおいて，次に来るべき命題であるとしよう．$p(i)$ が成り立ち，$p(j)$ と $p(i)$ の間の命令が実行されれば $p(j)$ が成り立つように，$p(i)$ と $p(j)$ の間のコードができていることを証明しよう．この検証過程をすべての制御の流れに対し，隣り合っているすべての命題の組に対して実施するならば，プログラムはもしそれが停止するとすれば，帰納法によって，正しいことが言える．この検証過程は**部分正当性**(partial correctness)

（すなわち，停止する場合に対して）だけを示しており，もう一つ停止性の証明を補なわなければならない．興味のある読者はこの技法の例を文献で参照されたい［Elspas ほか，1972；London，1970］†)．

検証過程の適用に対するいくつかのコメントは，大切であろう．第一に，プログラムが正しいと証明できなかったとき，プログラムは正しいけれども命題が誤っていたということもあり得る．それゆえ，証明の創造的な部分は命題を見つけることであるが，これはかなりの規模のプログラムに対しては非常に困難になる．プログラムの証明はまた非常に長くわずらわしいものになりがちである．そのような証明の正当性の検証の問題は，あきらかなものである．プログラムが正しいことを証明する努力は非常に大きいので，大規模なものに対してこの方法を適用することは，しばしばあきらめてしまう．最後に，この種の証明では，キーパンチのような事務的な誤りを防ぐことはできない．また，通常，証明では正しいと仮定しているハードウェアの誤りも防ぐことはできない．しかし，最も重要なことはソフトウェア・システム全体を検証のための一つのプログラムとして見做すことが，現在の状況では非現実的であるという点である．

命題を用いる方法は必ずしも証明であるとは宣言せずに，経験的に使用できるものであり，非常に一般的で柔軟な方法である．たとえば，ある言語（たとえば，SUE［Clark および Horning，1971］）においては，プログラマは命題を指定できるが，それはコンパイラがデバッグモードのとき，誤りチェックの命題を生成する．このようにして，それらの命題はある入力に対して検証される．コンパイラが“正規”モードのとき，同一の命題はコメントとして挿入される．同様の方式が会話型プログラミングに対して提唱されている．そのような環境の下では，プログラマは命題を作り出し，システムはそれを検証したり誤りであることを証明したりする．システムからのフィードバックは，プログラマが自分のプログラムの性質を理解するのに役に立つのである［Snowdon，1972］．

プログラムの正当性を証明することを，プログラムの設計や実現とは独立の活動としてこれまで議論してきた．これは実際とはかけ離れている．第一に，

† 訳注．プログラムの正当性の証明に関する形式的な議論については，たとえば，Z. Manna: Mathematical Theory of Computation, McGraw-Hill, 1974（訳書：五十嵐 滋訳・**プログラムの理論**，日本コンピュータ協会，1975）などを参照されたい．

誰か他人のプログラムが正しいことを証明するのは，自分自身のものを証明するよりもあきらかにずっと困難であろう．さらに，プログラムの構造や設計が正当性の証明の複雑さあるいはその可能性さえ大きく左右するのである．したがって，正当性の証明は独立に考えてはならない．むしろ，それは設計や実現の努力の一つの部分である．正当性の証明はプログラムの設計をある方向に持っていくかもしれない。設計とコーディングの際に，構造的プログラミングの技法とプログラムの証明の技法とを結合すれば，信頼性の高いプログラムを達成する最適な方法となるように思われる［Hoare, 1969］．この節と次節とでは，正当性の証明を設計とは独立に議論しているが，それらは設計と実現に含まれている部分であることを，読者は常に頭においておかなくてはならない．それは適当に強調するために，分離したにすぎないのである．

手で行なうプログラムの証明は，テストを補足しきれいなプログラミングの様式とプログラムの注意深い設計とを行なうことに影響を与える．後者は，それ自身，価値のある目的である．次節では，オペレーティング・システムに適用した場合の様に一般的な方法をもっと細かく論じてみよう．

9.4.2 システムの正当性

オペレーティング・システムは正当性を証明するのには大きすぎる．しかし，その性質，仮定，定理の精密な記述はその信頼性を大きく左右する．副次的な利益として，その正当性の証明を試みている間に，プログラムの動作の非常にくわしい理解が得られる．すべての必要な性質，命題，定理を形式的に述べたときには，システムは充分非形式的に解析してあるので，すべての実用的な目的に対して正当性を保証することができる．

さらに，システムの正当性の証明を行なうためには，入力やプログラムの環境を左右する他の要因に関する仮定の非常に厳密な仕様が必要となる．こうすれば不適当な起動によって生ずるシステムのモジュールの誤った操作の危険性を，最小限にすることができる．プログラムを実現する人が，最終的にはそれを正しいものと証明することを知っていれば，プログラムの様式や構造に対してよい影響を与えるであろう．それゆえ，現在の状況では大規模システムを完全に正しいと証明することは恐らくやめてしまうだろうが，正当性の方法を試

みたり計画したりすることは，それ自体有益な練習になる．

単純な順次型プログラムの正当性の性質は，入力から出力への関数として表現したり，プログラムの変数の間の直接的な関係として表現する。そのような性質は第 9.4.1 節に示したような検証過程を用いて，プログラミング言語の意味から多かれ少なかれ直接証明することができる．システムの正当性には，いくつかの問題点が追加される［Ballard および Tsichritzis, 1973］.

正当性の証明を試みる前に，正当性の性質を明確に言明しなければならない．システムの各モジュールが提供する機能の正確な仕様を与えることは，しばしば非常に困難なことがある．オペレーティング・システム全体が一つの単位として行なうことを指定することは，さらに難しいことである．たとえば，デッドロックの存在しないことなど，いくつかの明確な条件もあるかもしれない。しかしながら，実時間の制限など多くの条件は，形式化するのが容易ではない．たとえば必要な性質が理解できたとしても，入力と出力との間の関数として便利に記述することはできないことが多い．たとえば，大規模な共有データ構造を処理しているプロセスを考えよう．一つの重要な性質は，たとえば，結合リストを正しくつなげたままにしておくように，プロセスがそれを処理した後でもデータ構造を一貫性のあるものとしておくことである．一貫性は，ポインタの意味を用いれば表現できるであろうが，それを入力と出力との間の関数を用いて記述することは，かなり馬鹿げている．

大抵のシステムの操作には，本来，自然にまたは論理的に並列性が存在する．それゆえ，並列性を扱うことのできる技法が必要となる．時には，任意のタイミングの条件を調べなければならないが，それは証明すべき場合の数を大幅にふやす．

いくつかのシステムのプロセスは，概念的には永久に走っている．実際，オペレーティング・システムの一つの望ましい性質は，それが停止しないことである．うまい因果関係と関数を実現する順次手続きにおける CALL と RETURN 文の単純な制御は，オペレーティング・システムではいつでも存在するわけではない．むしろ，連続的に成長するプロセスが存在するが，それは内部状態の履歴を用いて記述できるのである．そのようなプロセスは入力の流れを受け取り出力の流れを送り出す．入力と出力の順序は，各プロセスにおける任意

の複雑なスケジューリング機構によって，交換されることもある．

あるプロセスは，ハードウェアに近い．ハードウェアの操作に関する概念的な困難さは何もないが，一般にそれを記述するのは面倒である．一つの理由は，ハードウェアは多くの変わった方法で誤りを犯すからである．これらの可能性は，実際には無視しているが，正当性の証明においては，それらを述べておかなければならない．ハードウェアの性質の精密な記述は，証明の基本的な部分として必要になってくる．

システムの正当性を証明する際の前述した困難さの大部分は，第2.2節に述べた状態の枠組［Horning および Randell, 1973］を採用すれば，解決することができる．システムは，任意の一瞬に，その状態によって表現できる．プログラムの意味は，ある状態変数の値によって指定するシステムの状態に対するプログラムの効果によって記述することができる．システムの進行はその状態の履歴によって表わされる．状態の枠組を用いれば，抽象化の任意のレベルにおいて，各プロセスを記述することができる．そうすれば，性質は状態間の関係として記述することができる．たとえば，順次型プログラムの検証において，状態は命題を指定している点に対応する．正当性の証明は実行の最中に従っている状態の系列に関して，帰納法を用いることによって行なう．

状態の枠組に関する問題は，それがこまかすぎるということである．証明を進めるためには，特定の状態または状態の集団に対して，特別な注意を払わなければならない．順次型プログラムの正当性においては，命題を選択するのは困難な仕事である．システムの枠組においては，興味のある状態の適当な系列を選択するのは非常に困難である．

システムとその状態の履歴を考えよう．正当性の性質は，通常，非常に高レベルの抽象化における状態または状態の集団の間の関係によって指定することができる．低いレベルでは，プログラムの性質（すなわち，プロセスが従うべき台本）は，コードとプロセスの制御の進行とに密接に関係している状態の関係として表現できる．正当性の証明は，異ったレベル間のギャップの橋わたしをすべきである．通常，それは4つのステップに従う［Ballard および Tsichritzis, 1973］：

1. 各プロセスの環境の性質を，明確にかつ形式的に述べる．通常，それらはプロセスを実行する機械の正当性およびプロセスの開始に関する仮定について述べている．
2. プロセスの活動に対応する低レベルの状態の系列の性質を，プログラムとプログラミング言語の意味とを用いて証明する．プログラムの構造やプログラミング言語の性質は，このステップの難しさを大きく左右する．
3. 適当な高レベル状態かつ（または）状態の集合が選択される．これらの状態のすべての可能な系列の性質を，ステップ2で導いた性質を用いて帰納法により証明する．帰納法自体は，常に直接適用できるわけではない．
4. 高レベルの状態の間の関係を用いて正当性の性質を形式化する．これらの関係は，ステップ3の定理から導かれる．ここではシステムの構造が重要である．それがよく構造化されていれば，含まれている性質を非常に簡潔に述べることができる．

プログラムおよびシステムの性質の検証は，多くの関心を集めてきた [Mills, 1973]．ソフトウェアの信頼性を強調する現在の状態からみると，プログラミングの努力をくわしく吟味することは，重要である．超信頼性のあるシステムにつながるように，プログラムの構成と検証との技法を開発することが目標である．現在の感じでは，構造的プログラミングとプログラムの正当性の方法とを結びつけることが適切なものであると思われる．Mills によれば，正しい技法は存在するが，人々はそれを信頼して適切に使用することをしないということである．Mills による相似を考えてみよう．5歳の子供は三目並べ（tic-tac-toe）のゲームを完全にプレイすることはできるが，彼等は自分のゲームは完全であるということは知らない．したがって，彼等は集中せず，またゲームに負けるときもある．7歳の子供は，自分の手が完全であることを知っている．したがって，彼等が興味をもたない場合を除けば，ゲームに負けることはない．プログラミングは，おそらく，5歳児が三目並べをするような方法をやめて，7歳児のような作戦を採用すべき時期に来ているということである．偉大なプログラムは，時間をかけそれを行なうことに集中するような人々によって書くことができるのである．

9.5 性能評価

計算機システムが複雑性を増すにつれて，それらを評価する困難さも増してきた．初期の頃のシステムにおいては，開発の努力の大部分は設計と実現とに注がれていた．性能の評価は，単にあとで考慮することであった．しかしながら，いろいろのハードウェアやソフトウェアの製品の増加に伴い，それらの多くは同じことをしていると主張しているが，メーカとユーザ両者共に設計の場面においても使えるようなシステムの性能を評価する合理的な方法を見つけ出すことに注意を集中してきた．

計算機システムの評価においては，ハードウェアとソフトウェアとの特性を区別することは困難なことである．システムの性能を判断する最初の試みは，ハードウェアの速度に関することに集中していた．そこではソフトウェアは小さな役割しか果していなかった．しかし，現代の多重プログラミング・システムにおいては，“裸の”機械すなわちそのソフトウェアのない機械の特性を知ることは，特別に役に立つわけではない．処理量とかターン・アラウンド・タイムとかいうようなシステムの特性は，ハードウェア自身の関数であるとともに，オペレーティング・システムや作業負荷の関数でもある．したがって，オペレーティグ・システムの設計法が，システムの性能の測定の問題を真剣に扱うことは適切なことであろう．

計算機システムを有効に評価するためには，その目的を明確に定義しなければならない．収集できるデータの量が本質的には無制限の数のパラメータを越えてしまうような環境においては，非常に詳細に行なうことによって，簡単にあふれてしまう．したがって，重要な属性に関する調査にだけ制限をしぼることが必要であり，評価の費用が合理的な限度以内に納まるように維持しなければならない．これらの重要な属性は評価の目的を理解しさえすれば，分離することができる．

特に，性能測定に対する3つの基本的な理由が存在する．すなわち選択の評価，性能の投影および性能監視である [Lucas, 1971]．**選択の評価** (selection evaluation) は実在するシステムをいくつか比較するときに用いることができる．通常の動機は，ユーザの要求に一番適合するハードウェアとソフトウェア

の組み合せを選ぶことである．これらの必要性から選択の規準の集合が決められている．そして，システムはこれらの規準に基づいて比較する．意味のある比較を行なうためには，各システムの原型は少なくとも一つ存在していなければならない．**性能の投影**(performance projection)は，一般に未だ実現されていないシステムに対して行なう．設計者はハードウェアに専念する前に，アイデアを試すために予測法を用いることができる．**性能の監視**(performance monitoring)はすでに動いているシステムの特性に関係する．局所的な構成の改善は，システムの非能率さを見つけ出したデータに基づいて，行なうことができる．

評価の目的を与えたとして，どんな属性がその目的に合致するのであろうか．重要な特性を選択することは，これらの特性を実際に測定することと同じ位難しいことである．実際に，ある種のデータを正確に集めることが困難であると，調査すべき属性の選択に影響を与えるかもしれない．また，いくつかの特性は評価の決定に関しては他のものよりもずっと役に立つであろう．決定要因は評価の処理に対して，どれだけのお金が使用できるかということであってもよい．第9.5.2節では，もっとも一般的に使用される計算機システムの性能の規準を論ずる．第9.5.3節では，そのような規準に対する実際の値を決めるときに使用する技法をいくつか紹介する．

9.5.1 評価の目的

前述した3つの評価の目的は，大ざっぱには機械に対する設計，購入，使用に対応する．個々の目的を詳細に記述するのに加えて，これらの型の評価がまだ開発中のシステムに対して，どんなに重大な影響を与えることができるかということも示そう．

システム作製中の各段階において，性能を予測できれば設計者のアイデアを実際に実現する前にテストすることができる．そのような予測は解析モデルかシミュレーション・モデル（普通はこちらが使用されるが）を用いて行なうことができる．モデルを用いてシステム全体に及ぼす影響を測定しながら，いくつかの設計上の選択を比較することができる．さらに，最終的なシステムの特性を，生産に入る前に決定することができる．好ましくないシステムの特徴は，変更が一番容易に行なえるような設計の段階において，修正することができる．

最終的なシステムの仕様を知ることにより，営業の人達はユーザにそのシステムを売ろうとする気持を持つようになる．

システムを設計している間にその性能を投影することにより得られるこれらの明白な利点にも拘わらず，過去においてはお金の問題によりそのような予測の効果を，しばしば減少させてきた．使用できる資金源の大部分は設計のために向けられているので，性能の予測は設計よりも数カ月遅れてしまうことが多い．有益な解析もそれがあらわれたときは，いつも遅すぎて役に立たない．予測が計算機システムの設計において有益であるならば，それはプロジェクトの他の部分とペースを合わせるように，充分高い優先度を与えておかなければならない．

あまり利用はされていないが投影の第二の特徴は，実在するシステムの修正におけるものである．また投影はシステムの作業負荷の変更の効果を決定することにも使用できる．システム管理者はそのようなデータをシステム構成の将来の変更の計画をたてるために使用することができる．

選択の評価は，ユーザの要求に一番適している構成を選択するために，いくつかの実在するシステムの解析を行なう．速度，信頼性，費用が，すべてそのような選択の対象になる．ユーザが重要だと思う属性の各々に基づいて，システムをランクづけすることができる．他の方法としては，**評価点**（figure of merit）と呼ぶ属性の組み合せを計算することもできる．この数は属性の荷重平均であり，システムを単一のランクづけするために用いることができる．最高の評価点を持ったシステムが，テストしたシステムの中では最高のものであると考える．

選択の評価は実在するシステムを拡張するときに，性能の投影と組み合わせて利用することができる．投影を用いれば周辺装置とかソフトウェアのパッケージなどを追加したときの効果が決定できる．それから異った製造メーカで作った装置が比較でき，その中で最高のものを選択することができる（すなわち選択の評価）．

監視はシステム設計の道具および性能評価の道具である．ユーザやシステムの特性の変化を見張ることは，効率よく操作するようにシステムを“調整”し続けるために役に立つ．それゆえ，良い監視機能は，どこの設備でも目的とすべきものである．それはまた選択や投影を支援する技法としても考えることが

できる．システムの動作に関するデータはそれをランク付けし，また負荷や構成が変更する際に，変化を予測するのに役に立つ．特別な監視の方法を第9.5.2節で論じよう．

システムの設計，選択，監視を助ける以外に，性能評価は将来のシステムの設計に影響を及ぼすかもしれない．システムの作業負荷やユーザの要求の特性を見つけることにより，設計者は性能評価を利用して要求に合致するようにシステムを仕立てることができる．信頼のおけない効率の悪い設計を見つけ出し，それを修正することができる．恐らく，評価の最も重要な長期的な利用法としては，システムの性能に対する標準を設定することである．保険業務機関のように，プログラム検閲当局は計算機ソフトウェアの性能に対する最小限のレベルを作り出すことをするであろう．しかしながら，そのような標準が確立される前に，どんなものが“よい”性能を構成するかということについて，もっと研究することが必要である．

9.5.2 性能の量

性能をあらわすのには3つの主要な量がある．すなわち処理量，ターン・アラウンド・タイム（またはレスポンス・タイム）および可用性である [Calingaert, 1967]．作業負荷の能力は**処理量**（throughput）すなわち単位時間当りに（平均的に）実行できる作業の量で測定する．**ターン・アラウンド・タイム**（turnaround time）は，一つのジョブがバッチ・システムを通過する平均時間である．**レスポンス・タイム**（response time）は，会話型システムがユーザのコマンドに応答するために必要な時間をさす．**可用性**（availability）は，一日のうちどれ位システムが有益な仕事ができるかということを示す量である．保守の時間と予期していないダウンの時間は，この量に入っている．時には，**平均故障間隔**（mean time between failure）を可用性のかわりに用いる．前者の方が，システムがそのユーザに対して何が出来るかということを定量化しているという点でユーザ向きである．これらの量はシステムが与える処理が，何故そうなるのかということを必ずしも指摘はしていない．

もう一つのクラスの量は**効率量**（efficiency measure）というが，これはシステムが内部的にどのように動いているのかを知るきっかけを与えてくれる．オ

ーバーヘッド，資源使用率，機械の速度などのような内部の特性は，ユーザの社会に対しては本来重要なものではない（恐らく，価格に対する影響についてはそうではなかろうが).実際，これらの量を最適化することは，ユーザにとっては有害になるかもしれない．たとえば，資源の利用率を増加すると，あるユーザにとってはターン・アラウンド・タイムが増加するかもしれない．しかし，そのような性能の属性はシステムの動作の非能率性を見つけ出すためには，設計者や管理者にとっては非常に有益である．

評価に役に立つ性能の量の種類はよくわかっているけれども，これらの測定をどのようにして正しく行なうのかは，常に明白であるわけではない．システムの内部特性が外から隠れているような効率量においては，特にそうである．これらの特性を決定するのに使用できる技法の種類については，次節の主題とする．

9.5.3 性能の定量化の方法

性能評価のための技法は，測定する属性や解析するために使用できる資源によって，大きく左右される．評価の方法は，３つの分野に分類することができる．すなわち，解析的，経験的およびシミュレーションを用いた方法である．解析的方式では，システムを数学的モデルとして，抽象化する．次に，このモデルを解析し，その結果をもとのシステムに適用する．経験的方法では，実際のシステムに関するデータの収集を行なう．次に，このデータは，直接引用する（たとえば，命令の実行速度）かまたは別の解析（たとえば，軌跡で駆動するシミュレーション）．のための入力として用いることができるシミュレーション・モデルは不必要な詳細な部分は取り除いてしまった，実際のシステムを抽象化したものである．このモデルに関してデータを収集することは，同様のデータをもとのシステムから収集するよりも容易であると考えている．そして，このデータをモデル化しているシステムの属性を記述するために使用する．

前述した技法は，いづれもすべての評価の問題に対して理想的なものというわけではない．しかし，各方法に対し，それがすぐれているような状況が存在する．以下の議論では，各技法の主要な能力，弱点，重要な応用について考えてみよう．

9.5.3.1 解析的モデル

計算機システムの評価に適用できる数学的モデルには，いくつかの種類がある．最もよく使われる方法の一つは，待ち行列理論である．この技法は，CPU および周辺装置の両方を含むスケジューリングの問題に適用されてきた．これらのモデルにおいては，タスクの到着時間に対する分布を決める関数を仮定する．処理装置に対するスケジューリング方式を選択する．次に，タスク毎に必要な時間など処理装置自身の性質を与えて，システムの特性を導く．たとえば，CPUで処理するプロセスの待ち行列においては，各タスクに対する待ち時間の期待値，各タスクを処理する時間の総計（すなわち，ターン・アラウンド・タイム），単位時間当り処理されるジョブの平均個数（すなわち，処理量）などを知ることは，役に立つであろう．待ち行列理論のモデルは，CPU，ドラム，ディスクのスケジューリングの解析に，特に成功をおさめてきた．Coffman [1969] や Fuller および Baskett [1972] は異ったスケジューリング方式とドラムの構成に対して，待ち行列の長さや待ち時間の期待値を解析してきた．ディスクのスケジューリングの方針は，平均待ち時間 [Turnbull, 1972] やシーク時間の期待値 [Teorey および Pinkerton, 1972] に関して，調べられてきた．待ち行列理論のモデルの入門書として，読者は Morse [1958] および Coffman と Denning [1973] の著書を参照されたい．

組み合せ理論的解析と数理計画法の技法も，システムの評価に適用してきた．組み合せ理論はいくつかのスケジューリングの方式の最適性を証明するために用いてきた [Sevcik, 1971]．線形計画法および動的計画法も，システムの解析のために用いてきた．このやり方は，一般に，評価の段階よりもむしろ設計の段階において適用されることが多い．しかし，これらの方法は，すべて重大な欠点を持っている．

解析的技法の鍵となる問題は，いくつかのサブシステムの複雑な相互作用をモデル化する能力に欠けるという点である．周辺装置のような単一のサブシステムにおいては，ある単純化の仮定をすれば解析が可能となる．しかし，これらの仮定はシステムをもっと全域的な視点から眺めると，正しくないことが多い．変数や確率分布の増加により，解析的な解を可能にしている単純な抽象化を破壊してしまうことが多い．

さらに，システムのある部分は数学的に表現することが困難である．大きなソフトウェアは理論的に解析しにくい．モデル化できるようなシステムの構成要素でさえ，パラメータの変化に過度に敏感になるような技法を生み出す．すなわち，構成要素の構造をちょっと変更しただけで，その技法の適用が可能でなくなってしまう．また，多くの単純化のための仮定の正当性が，疑わしくなってしまう．たとえば，多重プログラミングは処理装置のスケジューリング方針の関数であるような効果を生み出す．これらの効果は，通常，ランダムと考えられるが，経験的な証拠からこの仮定を支持するかどうかということは，常に明白であるとは限らない．しかしながら，これらの制限の大部分は全域的な状況に対して多く適用されるのである．小さいサブシステムの性能の評価においては，解析的技法は実在の機械に適用できるような正しい結果を作り出すことができる．さらに，何を測定し経験的データをどのように評価するかを決めるときには，これは非常に役に立つ．

9.5.3.2 経験的技法

恐らく，性能を評価するために最も広く使用される技法には，動作中のシステムからデータを収集することを含んでいる．これらの経験的技法は，2つのクラスに分類される．一つの方法の集団は，CPUのサイクル時間，命令の実行時間などのシステムの仕様を標準の負荷に適用することである．特に，命令の解析，命令ミックス，核プログラムはこの方法を用いている．第二の方法の集団は，実行しているプログラムからデータを収集することである．人工的あるいは実際のプログラムを，比較のデータを集めるために異った機械で実行する．

システム評価の初期の頃の方法では，命令のサイクル時間と加算命令の時間とを基本的な量として用いている．これらの時間は全体的な機械の速度について，ぼんやりした指標を与えるけれども，正確な基準としてはあきらかに不適切である．サイクル時間と加算時間だけを考えているのでは，機械の多くの特性が無視されてしまう．どの単一の命令をとってみても，命令の速度の代表とはならない．さらに，多くのシステムで限界を与える要因となる入力と出力とを，この技法を用いて考慮することは困難である．

サイクル時間や加算時間の方式の欠点は命令速度の荷重平均を用いることに

より，除くことができる．この平均は**命令ミックス**（instruction mix）と呼ぶが，平均的なプログラムで実行される回数に比例して，命令に重みを与える．異った応用に対して，異ったミックスが考案されている†)．科学計算用ミックスは計算用の命令に重きをおくが，事務計算用ミックスはそのかわりにデータ処理に関する命令に重きをおく．平均の命令の速度は機械の能力をあらわす量として用いる．

核（kernel）**プログラム**（注意：SUE 型の核とは，無関係である）は，機械の平均負荷の代表となるべき標準の関数を計算するプログラムである．このプログラムの各機械語に対し，わかっている命令の速度を適用して，時間を測定する．すなわち，各命令の実行の頻度によって荷重をつけた，すべての命令の総和が実行時間の推定値となる．核プログラムが命令ミックスよりもすぐれている主要な点は，システムの特別なプログラミングの特性を利用できることである．多アドレスやスタック命令を，核プログラムの実行時間を減少するために最大限に利用できる．命令ミックスでは少ない頻度でしか使用しないような多くの特徴が，核プログラムを用いたときには重大な影響を及ぼすかもしれない．この技法を適用するときは，大抵，広い範囲の計算問題を含むようにいくつかの核プログラムに対する荷重平均を用いる．

ハードウェアの速度を測定するほかに，核プログラムはハードウェアの微妙な隘路の評価やソフトウェアの評価に対しても，制限をつけて使用する．全体の実行時間に対し，どの命令が一番効いているかを観察することにより，設計者はどの命令速度を改良すれば，最も有効であるのかを知ることができる．また，異ったコンパイラによって生成される核プログラムのコードを調べれば，コンパイラの性能を大づかみに知ることができる．

ベンチマーク（bench mark）**プログラム**は，システムの典型的な負荷であるような，有益で一般的な関数を計算するアルゴリズムをコード化したものである．ベンチマークは，それが実際のプログラムであるという点で，核プログラムとは異っている．それらは本当の事務計算あるいは科学計算を計算する．一方，核プログラムは平均的な計算を代表すると思われる関数の一部分だけを

---

† **訳注．** 科学計算用としてはギブソン・ミックス，また，事務計算用としてはコマーシャル・ミックスなどが有名である．

計算すればよい．

応用の妥当な分布を実現するために，多数のベンチマークを一般に実行する．ベンチマークは，ここでは，入出力やシステム・ソフトウェアを考慮に入れた最初の技法である．ベンチマークは実際に実行するので，入出力の遅れやオペレーティング・システムのオーバーヘッドの効果は勘定に入っている．

システム・ソフトウェアを評価の対象に加えると，新しい問題が生じてくる．高速で最適化を行なうコンパイラを比較するのは難しい．同様に，遅いオペレーティング・システムは，速いものよりも多くの機能を提供することが多い．したがって，時間は必ずしも公正な比較の基準とはならない．また，ハードウェアとは異なり，ソフトウェアは一般に機械の存在する間中，変化する．製造メーカによるシステムの支援は，古いプログラムを改良し新しい機能をつけ加えることを含む．信頼性のないソフトウェア・システムが，およそ数カ月でその虫を修正し，以前には秀れていたシステムと肩を並べるようになる場合もあるかもしれない．ソフトウェアの改良に対して名声を持っている会社に対しては，評価も違ってくる．したがって，ソフトウェアの変更を可能とする柔軟性は測定するのが非常に困難なパラメータである．

合成（synthetic）**プログラム**はシステムの性質の多様性（たとえば，入出力，命令速度，オペレーティング・システム）を一様な方法でテストする（通常は，機械独立な）手続きである．合成プログラムが核と異なるのは，前者は実行するように完全にコード化してあることである．それらは実際の応用をあらわしてはいず，また，評価する以前には存在しなかったという点で，ベンチマークとは異なる．

合成プログラムは，実際のプログラムと非常に似たような振舞をするように設計している．それは計算をし，入出力をし，あるシステムの関数を呼び出す．ベンチマークよりもすぐれている点は，プログラムが単純で，容易に記述でき，持ち運びを可能にしておくことができるということである．

性能評価で最も頻繁に使用される道具は，間違いなくハードウェアとソフトウェアのモニタである．**モニタ**（monitor）は，通常のシステムの動作において，性能のデータを収集するプログラム，またはハードで固定した“暗箱”である．モニタは番地の軌跡，参照の回数，チャネルの使用率，記憶の使用率，ジョブ

の統計など，広い多様なデータを収集できる．監視することは，システムの内部動作を眼で見ることができるようにする，一つの経験的な技法である．

最近の計算機システムは監視機能をいくつか持っているけれども，ハードウェア・モニタは，一般にもとのシステムの付属物として作る．論理的には，どんな種類の情報の動きも，そのような装置によって測定できる．しかしながら，ハードウェア・モニタは収集するデータをかなり無差別に扱っている（たとえば，完全な番地の軌跡はほんの1分位の間に，テープ数巻になってしまう）．したがって，それを選択的に使用して，ある事象の標本だけが収集できるようにしなければならない．装置の使用率やアクセスのカウントは，ハードウェアの監視機構によって，最も容易に得ることができる．

ユーザ・プログラムやシステム・ルーチンの性質は，ソフトウェアによって一番良く監視できる．資源の使用とかシステム・ルーチンの呼び出しの頻度などのジョブの統計量は，プログラムのパッチを少しすれば，簡単に監視することができる．しかし，余分なソフトウェアによる監視は，CPU 時間や記憶空間を消費するので，システムの性能を低下させる．ハードウェアの監視は，通常，費用がかかるけれども，親のシステムに影響を与えることはない．システムの全生涯にわたり，ある種（たとえば，ジョブの実行時間）の監視は行なわれるので，ハードウェア・モニタの費用とソフトウェア・モニタによる性能の低下との兼ねあいを，考慮しなければならない．

注意深くソフトウェアによる監視を行なえば，システムの性能への影響は全CPU 時間の1％以下に制限できるという報告がある．残念ながら，多くのシステムは監視することを念頭に置いて設計していないので，監視することを容易には実施できない．たとえば，任意の通信路に小さい手続きを挿入できるような能力があれば，監視するのに有益な道具となる．

9.5.4 シミュレーション法

シミュレーションは．システムをテストしその設計の変更の効果を評価するためには，経済的に見合う方法である．システムの主要な要素を計算機のプログラムとして，モデル化する．すなわち，そのプログラムの実行は，システムの動作そのものをシミュレートしている．本質的には，2つの型のシミュレー

ションが存在する．すなわち，軌跡駆動型とモデル駆動型である．軌跡駆動型では，研究の対象としているシステムを監視して得られたデータを利用する．このデータは，シミュレータに対する入力として，システムの振舞いの標本を提供する．モデル駆動型シミュレーションでは，要素は確率関数に従って振舞う．たとえば，ドラム・チャネルのシミュレーションにおいて，アクセス時間は一様確率分布によってモデル化する（すなわち，モデル駆動型）か，または観察したアクセスの遅れの実際の軌跡（すなわち，軌跡駆動型）によってモデル化する．方法は，軌跡のデータが使用できるのかどうか，また確率関数によってどの位正確にシステムの振舞いがモデル化されるかということによって選択される．

シミュレーション法の背後にある仮定は，システムそれ自身よりもシミュレートしたシステムの方が，プログラムし易く監視し易いということである．平均的には，これは真実であるけれども，どちらの方法もとくに安いわけではない．シミュレーションの費用は，モデルの中でどれだけ細かくシステムを実現するかという点に左右される．より細かくすれば，より正確なモデルになるが，恐らく実行は遅い．実行時間を高速化するためには，正確さを犠牲にしてモデルを簡単化すればよい．さらに細かくしても，もはや費用の増加には見合わないという点を決めるのが，困難なことが多い．シミュレーション・モデルは統計的に正しくなければ，その結果を信用することができないのであるから，細かさの選択は経済的な問題だけでは決まらない．

シミュレーションの費用は実在するシステムの評価に対しては，はなはだ高価になることがほとんどである．しかし，設計の段階において，システムの性能を投影することは有益である．設計上の提案は，その有効性を決定するために，シミュレーション・モデルを用いてテストすることができる．資源配分の一つの問題は，設計プロジェクトの他の部門とペースを合わせることができるように，シミュレーション担当の人たちにも充分な資源を与えることである．シミュレーションの試みが失敗しているのは，要素の設計とそれをシミュレーション・モデルに入れるときの大きな時間遅れが，原因となっていることが多い．適切に使用すれば，シミュレーションは強力な評価の道具である．しかし，近い将来，解析的あるいは経験的な方法に取ってかわるほど，安くなり正確に

なるということは期待できない．

実用的価値のある２～３の解析的手法を除いて，性能評価の方法は，現在では科学よりもむしろ技芸である．大部分の技法は，その場限りのものである．正しいことが証明できる技法は，通常費用がかかり過ぎる．シミュレーションは，明きらかに有効であるが，その費用は大抵の計算機運用の枠を越えてしまう．

9.6 おわりに

第８章と第９章においては，良い品質のソフトウェアを設計し，実現することについて論じてきた．実際の場面では，概説した技法のうちのどれか一つまたはすべてを用いたとしても，最終的な製品は完ぺきなソフトウェア・システムとは，ほど遠いものであろう．“実際の世界”における最終的な信頼性や効率を保証するために，弱点を事実として認め誤りを予想しなければならない．システムの各モジュールとユーザとは，防御的に動作しなければならない．適当な運用の手続きが重要である．たとえば，情報の保全性は増分ダンプや完全ダンプを頻繁に用いることによって保つことができる．システムの鍵となる情報（たとえば，ディレクトリ）は，保護するだけでなく複写しておかなければならない．最後に，良い再開始機能があれば，故障の影響を最小限に押さえるがことできる．

ソフトウェアの製品は，完全に凍結されることはない．それは，修正を伴う生涯を過す．したがって，現実の姿は生産したソフトウェアの保守ということになる．機能の変更や誤りの発見は，新しいシステムのリリースを余儀なくされる．Belady と Lehman［1971］との興味のある研究によれば，モデルを使って保守の努力の増加率を探究している．ある条件の下では，保守の努力は時間と共に指数関数的に増加することが示されている．この結果は，実在するシステムに対する保守の努力の測定とも一致する．この事実から考えると，いくつかのソフトウェア・システムは，最終的には引退の運命にあるということがわかる．それを保守するよりも，書き直した方が簡単なこともある．これは，興味のある哲学的意味を持っている．ソフトウェアは，設計され実現される．そして（望むところは）それは生き，遂には死んでしまうのだろう．

演　習　問　題

9.1　システムの実現用言語として，PL/I の相対的な利点を論ぜよ．どんな望ましい特徴が，欠けているか．どんな特徴が，余分なのか．自分の場合を論ぜよ．

9.2　良いプログラミングの慣習を教える手段として，高速で診断のついたWATFIV や PL/C のような学生用のコンパイラを使ったとき，その利点と欠点について論ぜよ．

9.3　読者の同僚が書いたプログラムか文献に載っているプログラムをいくつか読み，各々の批判をせよ．自分が以前に書いたプログラムを読み，批評を書け．文献にあるプログラムで，優秀なプログラミングの例として指摘できるものがあるか．

9.4　新しい機械にシステムを作る（すなわち，システム生成をする）のに必要なステップを書き記せ．システムを開始する（すなわち，処理装置の初期ローディングをする）のに必要なステップを書き記せ．

9.5　一つのプログラミングの慣習として，プログラムを急いで書き，それから良いデバッギング技法や注意深いテストを用いて，誤りを見つける方法がある．この技法と構造的プログラミングとを比較して，その利点を論ぜよ．プログラミング技法を比較するための，実験の概略を述べよ．

9.6　**図2-7**にプログラムしてある生産者—消費者関係の正当性を，証明せよ．腕木システムの動作とその実現の性質について．厳密に定義しなければならない．この点において，プログラムが誤りを犯さないという保証を，どの位強く読者は与えるつもりか．

9.7　正当性の証明に対する，プログラムの構造の影響を論ぜよ．プログラムの構造と証明の構造との間には，何か関係があるのか．たとえば，London［1970］や Hoare［1971］の例を用いて，プログラムとその証明の例を考えながら，答えの論拠を示せ．

9.8 ソフトウェア工場の製品の検査の計画について，概略を述べよ．すなわち，信頼性，性能，ドキュメンテーションなどのような，プログラムの望ましい性質を検証するステップの系列を考えよ．

9.9 システムにおいて，最悪のハードウェアの故障は何か．そのような状況において，どのようにしてシステムの保全性の喪失に打ち勝つことができるか．

9.10 読者の意見で，頻繁に誤りを生じそうな PL/I の構成要素の例を見つけよ．その要素を除去したいか．それを他の要素と置き換えたいか．

9.11 ハードウェアの冗長性は，時には，信頼性を増すために使用される．冗長性の概念は，ソフトウェアの信頼性を増すという観点から重要であるか．例を示せ．

9.12 正しくないもの，構造の悪いもの，効率の悪いもの，これらの3つのプログラムの例を文献から見つけよ．そのような状況を修正する別のプログラムを提案せよ．

9.13 GO TO 文に対する必要性を取り除くために使用できる，制御文の集合を定義せよ．それらで十分であることを，証明せよ．それら一つ一つが必要であることを証明できるか．

9.14 ハードウェアから JUMP 命令を取り除くことを提案する人は少ないのに，どうして多くの人々が GO TO 文に反対をするのか．

9.15 新しいソフトウェア・パッケージを，受け取る立場にあると仮定しよう．そのパッケージが満して欲しい，実際的な性能の判定基準のリストについて概説せよ．そのリストと標準的なソフトウァアの契約とを比較せよ．

9.16 読者の施設で，処理装置は，X社，周辺装置はY社，オペレーティング・システムはX社が開発したもの，データ・ベース・システムはZ社が開発したもの，応用プログラムは自分で開発したものがあると仮定する．誤りを指摘し適当な保守チームに知らせるための手続きの概略を述べよ．

9.17 異ったシステム実現用言語について，比較研究をせよ．

9.18 性能評価のための道具のリストを作成せよ．それらの使用について，例を示せ．

9.19 設備の選択についての事例研究を準備せよ．たとえば，読者の大学において，CDC の設備に対し IBM の設備の利点を論ぜよ．

第 10 章

システムの例

10.1 はじめに

オペレーティング・システムの研究は，アイデアをそれ自身のために研究するような，理論的分野ではない．われわれはどの様に設計し，作り，それを使用するのかを学ぶためにオペレーティング・システムを研究するのである．個別の問題を研究するのに加えて，一つの完全なシステムにおいて，すべての要素をどの様にして結合するかを見ることは，重要なことである．プロセスの相互通信の基本命令，保護機構，記憶管理の構成，入出力システムは，真空中に設計するわけではない．これらの要素は相互作用をし，その相互作用を念頭に置いて設計し，使用しなければならない．この章では，本書でこれまで述べてきた主要な要素が，一つの完全なシステムの中でどの様に相互作用をしているのかを，読者の眼の前に浮かべさせるように2つのオペレーティング・システムを紹介している．

記述している2つのシステムとは，MITRE 社で開発した Venus オペレーティング・システム [Liskov, 1972 ; Huberman, 1970] と，Toronto 大学で設計した SUE の核 [Sevcik ほか, 1972] である．これらのシステムは非常に異った性格を持ち，かつ明瞭な構造を持っているという理由で選択した．これらのシステムは，規模，ハードウェアの構造，同期の方法，保護および目標範囲に関して，非常に異ったものである．しなしながら，両者のシステムは構造的方式で設計された．抽象化と仮想機械のレベルが全体のシステムの構造において，はっきりと見える．そのために，それらを簡潔にかつ透明に記述することが，容易にできる．

SUE の核は，IBM/360 のファミリー機械の上で実行するように設計した．SUE プロジェクトの一つの主要な目的は，よく使われている機械に対し，構造的システムを作りだすための現状の技法を適用することである．これを書いている時点では，システムは完全に設計され一部分は実現されている．

Venus システムは，小型のマイクロプログラム可能な機械，Interdata 3 に対する完全な会話型システムである．この機械は大規模で高速で，高価なIBM/360 シリーズとはほとんど似ていない．実際，それは全く正反対である．SUE プロジェクトとは異なり，Venus グループは自分自身のハードウェアの特徴を，マイクロプログラムを用いて定義することができる．自分自身のハードウェア構造を定義できる能力は，オペレーティング・システムの設計にとっては非常に役に立つ．これはSUE とは対照的であるが，SUE では360 のハードウェアの要素の特異性により，オペレーティング・システムをきれいに構造化することが困難な仕事になってしまう．

表 10-1　SUE と Venus の比較

特　徴	SUE	Venus
ハードウェア	高速，大型，高い	低速，小型，安い
ハードウェアの構造	固定した命令セット	マイクロプログラム可能
記憶構造	ベース・レジスタを介した静的再配置	セグメント化およびページ化
システムの目標	多重プログラミング・システムの核	完全な会話型システム
プロセスの同期	ファシリティ	腕　木
入出力ハードウェア	ハードウェア・チャネル，制御装置，機器	マイクロプログラムによるチャネル，テープとディスクに対するハードウェア・チャネル
保　護	資格に基づいた保護	保護機構なし
料金計算	資　格	マイクロプログラムが，基本的なプロセスの統計を管理する．
提供される機能	ディスク・ファイル・システム テレタイプ入出力システム 資源マネージャ 多数のサブシステムの接続	テープ・ファイル・システム テレタイプ入出力システム 共通辞書による共有セグメント アセンブラ，エディタ，デバッガ

Venus と SUE の両者とも，プロセスを基本的な設計の単位として用いている．郵便受けを最初に用いようと試みてから，SUE グループは，最終的には，高度に構造化した“ファシリティ”と呼ぶ通信機構を用いることに決めた[Sevcik ほか，1972]．Venus における同期は，腕木によって行なう．2つのシステムは，同期に対する高レベルおよび低レベルの基本命令を用いるよい例を提供してくれる．

Venus および SUE は，他のいくつかの点で異っている．Venus のユーザは，本質的に全体の機械および記憶空間を共有しているので，保護が困難になる．一方，SUE のプロセスは資格の機構により，高度に保護されている．SUEは資格に基づいたディスク・ファイル・システムを提供している．一方，Venus は保護のないテープ・ファイル・システムを提供している．実際，構造的システムを打ち建てる基本的な設計方針以外には，SUE と Venus とは共通なものが非常に少ない．2つのシステムの主要な特性は**表 10-1** に要約してある．

この章の残りの部分では，システムの記述を行なう．SUE は第 10. 2 節に，Venus は第10. 3 節に述べる．最後の節では，他のいくつかのシステムについて興味のある読者のために，文献を参照しながら簡単に紹介する．

10.2 SUE システム

10.2.1 はじめに

SUE とは IBM/360 のファミリー機械に対し，Toronto 大学で設計されたオペレーティング・システムの核の名前である．SUEは OS/360 のような意味では，完全なシステムではない．むしろ，それはオペレーティング・システムの**核**である（第1. 2 節参照）・単純なディスク・ファイル・システム，資源割り当て，保護，料金計算などの構造，タイプライターの入出力システム，非同期のプロセスの生成と消滅などの機能が用意され，SUEの下で走るようないくつかの保護した独立なサブシステム（たとえば，テキスト・エディタ，バッチ・システム，会話型システムなど）を作り出すための基本的な道具として提供される．SUEの核の記述は，2つの部分に分けられる．10. 2. 2 節では SUE におけるプロセスの通信と資源の割り当てとを概説する．あとの節では，SUEがそ

のユーザに提供している実際の機能を述べる.

10.2.2 SUE の構造

10.2.2.1 プロセスとその通信

SUEの核は自分自身の処理装置の上で実行する非同期のプロセスの集団から構成される. 最初の一つを除いて, すべてのプロセスは他のプロセスによって生成される. したがって, 各プロセスは**親** (father) と呼ぶ生成者を持ち, それが生成した**子** (son) と呼ぶプロセスをいくつか持つことができる. プロセスの順序づけは, **生成木** (creation tree) と呼ぶ有向グラフによって定義する (**図 10-1**). プロセスAがプロセスBの親であるとき, またそのときに限り節点Aから節点Bへ孤がつながる. 生成木はシステムにおける基本的な構造として働く. 特に, それはプロセス間のすべての可能な通信路を指定する.

SUE におけるプロセスの 通信機構は, ファシリティと呼ぶ Hoare のモニタを修正した形のものである. **ファシリティ** (facility) はファシリティ・コールによって, その子孫から処理の要求を受け付けるプロセスである. ファシリ

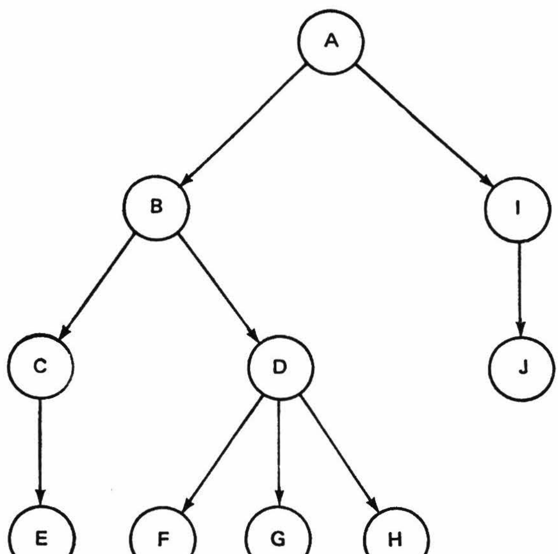


図 10-1　SUE-型生成木. (AがBの親であれば, 節点Aから節点Bへの弧が存在する)

ティは，3つの基本操作すなわち *Facility Call*, *Accept Facility Call*, *Complete Facility Call* によって実現する．Facility Call 操作は，その祖先（すなわち生成木における親，祖父，曽祖父など）の一つが提供している処理を，呼び出しプロセスが要求するために用いる．呼び出しプロセスはファシリティに対し，いくつかのパラメータを受け渡すことにより，処理の要求の性質を指定する．プロセスがこの基本命令を呼び出したときは，その実行は，一時，中断する．それを再開するのはその処理の要求が完結してからである．

各ファシリティは一つ以上の異った型の要求を処理するが，その型は呼び出しパラメータで指定する．処理の要求は Accept Facility Call 基本命令によって認められるが，それはペンディングされているある型の次の要求（すなわち，その型の最も古い未だ受けとられていない Facility Call）を Accept するといっているファシリティに対して，知らせるという効果を持つ．もしもそのような要求がなければ，そのような要求があらわれるまでそのファシリティは停止している．パラメータを変えることにより，ファシリティは Accept を“条件付き”にすることができるが，その場合待っている要求がなくても，停止はしない．Facility Call を受け取った後では，ファシリティは処理の要求に応じてある動作を実行する．ファシリティは，それから Complete Facility Call 基本命令を用いて，呼び出しプロセスの実行が継続することを許可する．

SUEにおいては，仮想機械のレベルを実現するためにファシリティ機構を用いている．生成木の各プロセスは，その子孫に対してファシリティを提供できる．たとえば，**図10-1**のプロセスBが“ファイル・システム・ファシリティ”を提供していれば，Bのすべての子孫（すなわち，C，D，E，F，G，H）は，Bによって処理される Facility Call を呼び出すことにより，ファイル・システムを使用することができる．**プロセス** $C, D, \cdots, H$ はファイル・システムを含むような仮想機械の上で，実行しているものと見なすことができる．プロセスが Facility Call を呼び出したときは，どのファシリティがその呼び出しを処理するのかは知らない．その仮想機械はそのファシリティを提供してくれるということを知っているだけである．したがって，各プロセスはその祖先が提供してくれるようなファシリティによって完全に定義される仮想機械を用いているのである．**図10-2** は SUE の核における実際の生成木を示している．特

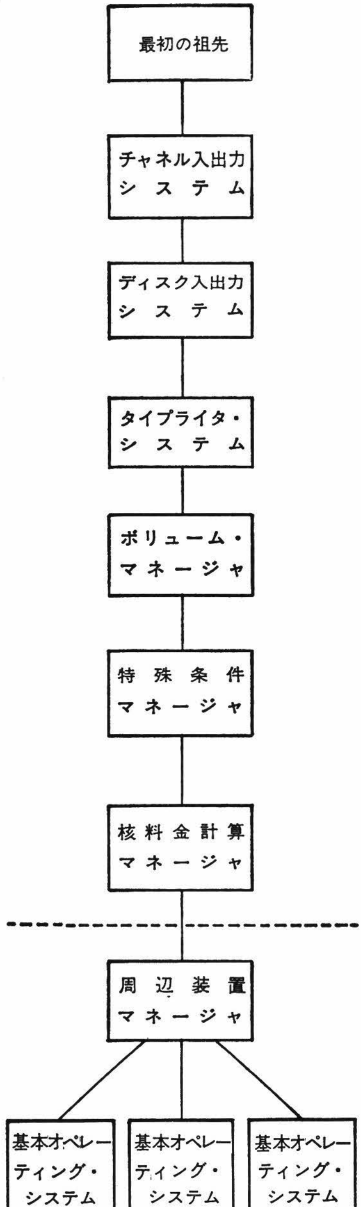


図 10-2　SUE における生成木の構造.（破線部分は SUE の核との境界を示している）.

定のファシリティの機能は，後の節で定義する．

上で議論したファシリティは**CPU プロセス** (CPU process) と呼ぶ．CPUプロセスのほかに，SUEには2つの型の類似プロセスすなわち**カレンダ時計プロセス**（calender clock process）と**バッファ・プロセス**（buffer process）とが存在する．バッファ・プロセスも，カレンダ時計プロセスも，処理装置の時間は消費していない．すべての CPU プロセスは3つのどの型の子でも，生成する権利を持つ．ただし，記憶とか処理装置の速度などいくらかの資源を，生成した子に解放しなければならない．生成されたときは，プロセスはその型によって次のような機能を持つ：

1. カレンダ時計プロセス　このプロセスは，本質的には指定した時間間隔が経過したならば，その親に対して Facility Call を行なう目覚し時計である．生成されると，そのプロセスはその親から最初の時間間隔を受け取るために待ち状態になる．

2. バッファ・プロセス　CPU プロセスはバッファ・プロセスの子を作って，その Facility Call をバッファに入れることができる．バッファ・プロセスはその親に対し Facility Call をし，それはさらに特定の祖先に対し Facility Call を行なう．バッファ・プロセス（親ではない）はその処理の要求が完結するまで待たなければならない．Facility Call が完結すると，バッファ・プロセスは再び親を呼び出し，バッファに入れておいた Facility Call が完結したことを知らせる．

バッファ・プロセスは郵便受けシステムにおけるメッセージ・スロットにきわめて似ている．CPUプロセスがファシリティを呼び出すとき，子のバッファを生成する（すなわち，メッセージを送る）ことができ，それが呼び出しを実行する．指定した Facility Call が完結した後では，バッファ・プロセスは，再び親を呼び出す（すなわちメッセージに答える）．その中間において，バッファ・プロセスは呼び出した祖先によって処理が行なわれるのを待っているが，（返事を待っている）バッファの親は別の仕事を続けることができる．

3. CPU プロセス　CPU プロセスはいろいろ変化に富んだ順次型プロセスである．CPUプロセスはシステムの中で，自分自身をファシリティ・プロセス

と定義することができるが，その場合ファシリティの木における位置に従ってFacility Call を処理することが許される.

10.2.2.2 資源，資格，保護

SUEシステムの内部での資源への要求は，資格によって管理する（第7.2節参照).資格は64ビットのレコードで，プロセスに対してある資源が使用可能かどうかを指定する．各プロセスに対応して資格のリストを与えてあり，それにより祖先が提供した資源のそのプロセスに対する割り当てが決まる.

すべての資格は，一般にその資格をもともと生成したファシリティかまたは核心（次節参照）が管理する．資格は陽にはその資格の管理者によって，また陰には，Complete Facility Call 基本命令の中で，ファシリティによって変更することができる.

各資格は，その管理者を指定する項目と次のような能力を支配するフラッグの集合とから成る：

1. 新しく生成した子に，資格を受け渡す.
2. Facility Call が完了したときに，資格を受け渡す.
3. 資格を生成したときに，それを調整する.
4. 管理者に知らせずに，資格を消滅する.

資格に対しては3つの異った形式がある．すなわち論理型，数値型および自由型である．その形式に応じて，資格の残りのビット（管理者の指定欄とフラッグのうしろのビット）は，次のように解釈する：

1. **論理型**　この資格は資源に対して制限したアクセスを行なうために，マスクとして解釈する．たとえば，ファイルの資格においては，一つのプロセスは読み-書きアクセスをするのに対し，別のプロセスは読みだけのアクセスをする.

2. **数値型**　この資格は整数として解釈する．数値型の資格は，一般に残っている CPU 時間とか入出力操作の回数のように，消費型の資源のために使用する.

3. **自由型**　この情報は，この資格を生成したプロセスにしか意味を持っ

ていない．

資格は，保護および資源の管理の両方に対して役に立つ．システムはほとんどすべての資源は資格として表現できるので，資格の操作すなわち *Create Capability*, *Inspect Capability*, *Modify Capability* によって一様な方法でそれらを生成し，処理し，消滅することができる．資格はまた安全でもある．一つのプロセスから別のプロセスに対し，資源の情報を転送する（Facility Call において，パラメータとして資格を用いる）ときは，資格リスト上における資格のスロットの番号だけしか受け渡しすることができない．資格リストは他のプロセスが資格を“偽造”できないように，プロセスから保護している．一つのプロセスは資格のスロットの番号を受け渡すことにより，アクセス権を別のプロセスに教える．第二のプロセスは，そこでスロットの内容を調べることができる．核心だけが資格リストを修正できるので，他のどんなプロセスも資源の割り当てについて嘘をつくことはできない．

資格はまた資源の使用の料金計算の機構も提供する．プロセスはそれが生成されたとき，その資格リストを介して，親から最初の資源の割り当てを受ける．その割り当てに対して責任を持たされたプロセスは，自分でその資源を消費するかまたはその子に受け渡すかする．各プロセスが割り当てられた資格のレコードを保持していれば，すべての資源の完全な料金計算が各プロセスの資格リストを調べることによって可能となる．

10.2.2.3 核　心

プロセス，ファシリティの基本命令，資格は IBM/360 のハードウェアに本来備わっているものではなく，**核心**（kernel）と呼ぶ第一のレベルのソフトウェアで実現している．核心は**図10-2**に示したファシリティの様な意味でのプロセスではない．むしろプロセス，割り込み，記憶，資格，ファシリティ，周辺装置，実時間時計などを管理する基本命令を備えることができるように，基本的な IBM/360 の機械構造を変更するようなプログラムである．核心が提供する仮想機械は IBM/360 の非特権命令のセットと核心用の基本命令とを持っているが，すべてのプロセスが実際に見えるのはこの機械である．核心はハードウェアを変更して SUE の核をプログラムするのにずっと便利な仮想機械を生

成しているものと見倣すことができる．読者はこの仮想機械と，Venus のマイクロプログラムが提供する仮想機械とを比較すべきである．

核心は記憶キーが0（すなわち，すべての記憶にアクセスできる）で，割り込みを禁止した特権モードで実行する．実際，これらの性格を持ったソフトウェアは，システムの中ではこれだけである．核心は本質的には機械全体を何の制限もなく使用できる．したがって，どのプロセスも核心から隠れることはできない．また，どのプロセスも核心の記憶の記憶キーを持って実行することはできないので，システムの他の部分からは核心を安全に保護している．それゆえ，どのプロセスも意図的にあるいは偶然に核心をこわすことはできない．

核心は核の残りの部分に対して，7つの機能を提供する：

1. プロセス 核心はプロセスの生成，消滅，実行の開始，実行の停止をする基本命令を提供する．各プロセスに対し，プロセス・ディスクリプタと資格リストとを核心の中に割り当てる．プロセス・ディスクリプタはプロセスそれ自身またはその親が調べることができる．生成されたときは，プロセスは停止の状態になっている．それは親によりいつでも開始できる．一旦，開始されると，プロセスは自分自身の仮想機械の上で実行する．割り込みは核心ですべて処理するので，プロセスには見えない．プロセスはそれが停止するかトラップされるまで実行を続ける（他のすべてのアクティブなプロセスと共に時分割される）．

2. トラップ（trap） 例外条件が発生したときは，その条件に付随していたプロセスは“トラップ”される．トラップはそれによって影響を受けたプロセスを停止させ，核心はその親に対して問題が生じたことを知らせるメッセージを送る．たとえば，プロセスは周辺装置からの割り込みを待つことができる．その割り込みが生ずると，プロセスは核心によりトラップされる．その親はプロセスに知らせるための手段を講じ（通常，ディスクリプタを変更する），それを再開始する．大抵のトラップは予測している．すなわちそれらを処理するために各ファシリティに特別な手続きを用意している．

3. ファシリティ 核心は第10.2.2.1節で述べたように，ファシリティと3つのファシリティの基本命令を実現する．

4. 資格 各プロセス・ディスクリプタに付随して，そのプロセスが持っている資格の完全なリストがある．Create Capability 基本命令はプロセスが新しい資格を作り出すことを許しているが，それは資格リストにつけ加えられる．Inspect Capability および Modify Capability 基本命令は，実在する資格を処理するために用いる．資格の形式は前節に述べた通りである．

5. 記憶管理 核心は記憶管理に対して，3つの基本操作を提供する．*Set Process Key* は，プロセスが実行すべき記憶保護キーを設定する．当然，そのプロセスはこのキーで実行するような資格を持っていなければならない．*Set Storage Key* は，そのプロセスが所有しているかまたは使用可能である2 K-バイトのブロックの記憶キーをリセットする．*Create System Area* は，プロセス・ディスクリプタに用いる記憶のブロックを割り当てる．核心はシステムの各プロセスに対しディスクリプタの領域が必要であるから，プロセスは核心に対し，それが生成した各々のプロセスのディスクリプタの記憶を提供しなければならない．記憶ブロック（それは資格の中で料金計算をされている）は資格の基本命令を用いてプロセスの間で転送することができる．

6. 時計 **カレンダ時計** (calender clock) は一日の日付と時間とを持っており，**実行時計** (run clock) は各プロセスに対し使用した処理装置の時間を持っているが，それらは核心が管理する．これらの時計は，特定の特権的プロセスが初期設定し検査する．

7. 入出力管理 周辺装置に対するすべての入出力操作は，核心の中で管理する．2つの基本命令すなわち *Request I/O Operation* と *Request I/O Status* により，チャネルを起動しチャネルの状態を検査する．これらの操作の詳細は IBM/360 ハードウェアに非常に依存しているが，ここで述べる範囲を越えてしまう．

10.2.3 SUE で提供する機能

第10.2.2節では，SUE の第一のレベルのソフトウェアすなわち核心について紹介した．核心の仮想機械を用いて，7つの非同期プロセスのシステム（**図10-2**）が核の基本的な機能を提供する．これらのプロセスの概略をこの節で述べるが，それは SUE の核が提供する処理の簡単な仕様を与えている．

10.2.3.1　最初の祖先

このファシリティは，SUE の生成木の 根に存在する．チャネル入出力システムのトラップを処理する（そのプロセスの親だから）のに加えて，最初の祖先は**主コンソール**（Archconsole）と呼ぶシステムの主要なオペレータ端末を実現する．すべてのプロセスは最初の祖先の子孫であるから，このファシリティの呼び出しを使えば，オペレータ・コンソールと“話す”ことができる．

10.2.3.2　チャネル入出力システム

生成木における第二の主要な構成要素はチャネル入出力システムであるが，それは仮想的な入出力機器の制御装置を生成する責任がある．すなわち，制御装置使用中のために生ずる遅れ，チャネルのエラー，予期しない割り込みなどは，このファシリティのユーザには透明である．このプロセスが提供する基本命令は *Execute Channel Program* と呼ぶが，それは核心に対する Request I/O Operation を用いて，与えられたチャネル・プログラムの開始を行なう．

チャネル・プログラムは，一般に，完了するのに長い時間がかかるので，プロセスがその間に何か役に立つことをしようと思えば，子のバッファを通じて Execute Channel Program ファシリティを呼び出す．

10.2.3.3　ディスク入出力システム

ディスク入出力システムは簡単なファイル・システムと外部（すなわち，SUE の形式に合っていない）ディスクとのインターフェイスを提供する．すべてのディスクへの要求は，それらが SUE の標準ファイルであろうと特別な形式の外部のボリュームであろうと，このプロセスを通して処理しなければならない．プロセスがファイルにアクセス（すなわち，読み込むか書き出すか）しようとするときは，3つの資格を提示しなければならない．第一の資格は，読み込まれたり，書き出されたりするコア領域の記憶キーの所有者であることを証明する．第二の資格は，そのプロセスが，そのファイルの正規のユーザであることを識別する．第三の資格は，プロセスに割り当てられたディスクの操作の回数のカウントであるが，入出力操作が実行されるごとに一つづつ減らす．

すべての SUE ファイルは連続した2 Kバイトのブロックから成り立っている．ディスクに対する読み書き操作は，常に，このような一様な大きさのいくつかのレコードに対して行なう．ファイルを所有しているプロセスは，そのフ

ァイルの領域内の連続したレコードの集合に対する新しい資格を作り出すかもしれない．このファイルの操作は“サブセット化”するというが，それによりプロセスのファイルの一部をその子孫と共有することができる．しかし，ディスク・システムは各ディスクのどの部分を誰が所有しているのかというレコードは持っている必要はない．プロセスがあるレコードのブロックに対して正しいファイルの資格を示せば，それらのレコードにアクセスすることが許される．それゆえ，ファイルをサブセット化したプロセスは，誰がそれに対するアクセス権を持っているのかを知っておく責任がある．

自分のディスクの形式を作りたいと思うプロセスは，ディスク・システムに対し自分自身を 識別しなければならない．そこでディスク・システムは SUE のものではないような形式でそのディスクに書き出すことを認めるような資格をそのプロセスに対して作ってくれる．

最後に，ディスクの入出力システムは論理駆動装置を起動したり，停止したりする．これらのコマンドは主としてボリューム・マネージャが使用して，ディスク駆動装置を駆動したり停止したりする．

10.2.3.4 タイプライタ・システム

タイプライタ・システムはプロセスがタイプライタ端末を介して外界と連絡するときのインターフェイスを提供する．ユーザは端末から登録し連絡しようと思うプロセスの名前を指定する．端末と通信を望むプロセスは，タイプライタ・システムに対し Facility Call を行ない，番地づけに用いる名前を定義する．そこで，その名前を用いて登録してきた最初のタイプライタに対しそのプロセスを割り当てて，Facility Call を完結する．話をすべき端末を，一旦，割り当てられれば，プロセスは一般にバッファとして2つの子を作る．一つは，メッセージを端末とやり取りする．もう一つは，緊急信号が来たときプロセスに警告を発する．入出力は実際にタイプライタ・システム自身によって開始される．

10.2.3.5 ボリューム・マネージャ

ディスク・ボリュームのマウントやディスマウントは，ボリューム・マネージャが監督する．プロセスはボリューム・マネージャに対する Facility Call を行ない，ディスク・パックのマウントを要求してもよい．そのプロセスは要求したボリュームやマウントしてほしい論理駆動装置の所有権を示すために，資

格を提示しなければならない．各ボリュームは一意的な6文字の名前で識別するが，ボリューム・マネージャはそれを1バイトの番号に変換する．この番号はボリューム・ラベルの内部識別名として働く．呼び出しプロセスの資格をチェックした後で，ボリューム・マネージャは適当なタイプライタ端末（すなわちオペレータ・コンソール）にメッセージを送り，そのボリュームをマウントすることを要求する．ボリュームがマウントされると，ディスク入出力システムに対し Facility Call を行ない，駆動装置が起動することを要求する．ボリュームのディスマウントも同様の方法で行なわれる．

10.2.3.6　特殊条件マネージャ

特殊条件マネージャは最初の祖先といくつかの外部機器との結合をする．最初の祖先からのコマンドに対し，特殊条件マネージャはタイプライタ端末と（タイプライタ・システムを介して）接触を行ない，それが主コンソールであることを宣言する．重大なシステムの故障があった場合，死んでしまう前に最初の祖先は特殊条件マネージャにいくつかのレコードをディスクに書き出すことを要求する．また，料金計算用の情報も当然のこととして同様の通信路を介して記録する．

この（すなわち，最初の祖先から特殊条件マネージャへの）通信路では，プロセスがその子孫の処理を呼び出すことが許されていないという意味で，システムの構造に違反している．実際，これはそのような違反の唯一のものである．その理由はあきらかである．生成木の根にいる最初の祖先は他のどんなプロセスに比べても，システムのことをよく知ることのできる位置にある．しかし，その特異な位置を利用するためには，生成木の下の方にあるディスク入出力システムとタイプライタ・システムとに連絡しなければならない．システム構造に違反することにより，特殊条件マネージャは最初の祖先から他のプロセスに対し，必要な通信路を作り出している．システムの構造におけるこの特異性は，デッドロック状態が生じないように特別注意して扱わなければならない．たとえば，最初の祖先が特殊条件マネージャを呼び出したとき，今度はそれが最初の祖先を呼び出す（すなわち，両方のプロセスが永久に待つ）可能性がある．

10.2.3.7　核料金計算マネージャ

核料金計算マネージャは，SUEの核の下で走るサブ・オペレーティング・シ

ステム（たとえば，バッチ・システム，会話型システム，テキスト・エディタなど）の動作を監督する．その2つの主要な機能は，アクティブなユーザに対しシステムの資源を割り当てることおよびサブ・オペレーティング・システムのオペレータ・コンソールとして働く核コンソールを実現することである．

ユーザのサブ・オペレーティング・システムを開始するためには，核コンソールのユーザは，タイプライタ端末から登録し核料金計算マネージャと会話することを要求する．通信を確立した後で，核コンソールのユーザはある集合の資源（たとえば記憶，処理装置の速度，ディスクの駆動装置など）を自分のシステムに割り当てることを要求する．これらの要求がこのサブシステムに許されている最大値を越えずまた資源が現在使用可能であるならば，サブシステムは開始される．そうでなければ，核料金計算マネージャは問題の核コンソールに知らせ，要求した資源を減少させるように要求する．

開始した後で，核コンソールのユーザはサブ・オペレーティング・システムの主オペレータ・コンソールになる．そのユーザは状態をきいたり，資源割り当てを変更することを要求したり，そのシステムと通信したり，システムを終了したりすることができる．サブ・オペレーティング・システムの資源の使用状況は，核料金計算マネージャによってディスク・ファイルに格納し，請求書を作るときに後ほど使用することができる．

生成木の核料金計算マネージャの下に，カード・リーダやライン・プリンタのような周辺機器を管理するファシリティを持つことは，適当であろう．周辺機器のマネージャは，SUE の核には含まれてはいないけれども，それは核の外側の生成木の上にのっている最初のファシリティになると思われる．

10.3 Venus オペレーティング・システム

Venus オペレーティング・システムは，Interdata 3 という小型で低速でマイクロプログラム可能な機械に対し MITRE 社が開発した，会話型の多重プログラミング・システムである．Venus 機械は Venus マイクロプログラムをかぶせた Interdata 3 であるが，ページ化とセグメント化の機構を持っている．各セグメントには手続きかデータが存在するが，それはシステム内の任意のユーザ（典型的には，3つのユーザが存在する）からアクセスする．操作する前に，

ユーザは一つ以上のカード・デックをシステム内に読み込ませ，セグメントとして格納しておく．そして，ユーザはテレタイプの端末から登録をする．端末から呼び出すコマンドにより，磁気テープからセグメント内にファイルを読み込むことができる．他のコマンドにより，一つ以上の手続きのセグメントを実行することができる．手続きの実行はデバッギング・モードの下で行なってよい．プログラムをテストするために，ユーザはプログラム内に“ブレイク・ポイント”を指定することができる．すなわち，ブレイク・ポイントの位置に達するとデバッギング・ルーチンに制御が移り，そのユーザは自分のプログラムの状態を会話形式で調べることができる．終了した計算の結果を出力セグメントに格納し，今度はそれをラインプリンタに掃き出すか，磁気テープに貯えるかしておく．

Venusの主要な設計目的は，“良い”ハードウェア・アーキテクチャがあれば，ソフトウェア・システムの設計がずっと簡単になることを示すことであった．この目的のために，機械の構造がシステムの構造に影響を与えるようにした．実際，機械はマイクロプログラム可能なので，設計者はその必要性に応じて機械の性格を鋳造できる．明確なハードウェアの基礎を与えて，仮想機械のレベルを追加し，誤りを少なくして構造的システムを作ることができる．

10.3.1　ハードウェア

Interdata 3 は2000命令分のマイクロ記憶を持ったマイクロプログラム可能な小型の機械である．Venusシステムは64バイトのコア記憶，磁気テープ駆動装置2台，（ページ化に対する）0.5M（メガ）バイトのディスク，テレタイプ数台，ライン・プリンタ1台，カード・リーダ1台から成る構成をしている．主記憶にアクセスできるハードウェアのチャネルがあり，一つはテープ駆動装置へ，もう一つはディスクにつながっている．他の装置に対するチャネルは，マイクロプログラムによりシミュレートする．命令サイクルの間に，マイクロプログラムは要求のある任意の装置（テープとディスク駆動装置を含む）の処理を実行する．したがって，ソフトウェアは，通常入出力割り込みを処理するために必要な実時間の制限からのがれることができる．チャネルのインターフェイスは第10.3.5節でさらに議論する．

外部装置の終了によって割り込みが起きないので，Venus システムの低レベルにおいては，複雑な帳簿づけの雑用から解放される．入出力割り込みを持っているシステムにおいては，入出力終了割り込みが誰に属しているのかを決定することが困難なことが時々ある．たとえば，そのようなシステムにおいてディスクの割り込みが生じたとき，そのディスクをどのプロセスが起動したのかを，ソフトウェアの“割り込み処理システム”が見つけ出さなければならない．いくつかのプロセスがディスクを共有していると，これは簡単なことではないであろう．この種の問題は，Venus では決しておきない．すべての割り込みは算術上のあふれ，不当命令の実行などの条件として**実行中**（running）のプロセスに対応させる．したがって，どのプロセスが割り込みをひき起したかということに関しては何のあいまいさもない．

10.3.2 プロセス

Venus マイクロプログラムは16個のプロセスを実現する．各プロセスはコアに常駐の作業領域と機械のすべてのセグメントにわたる番地空間とから定義される．作業領域は汎用レジスタ，プログラム・カウンタおよび他のプロセスに関係する情報（たとえば，腕木待ち行列に対するリンクとかソフトウェアで定義した優先度）などが含まれている．

プロセスの間の通信は腕木によって行なう．各腕木に対し腕木の値とその腕木に対する停止リストの最初のプロセスに対するポインタから成るヘッダ要素が与えられている．与えられた腕木に対して停止しているプロセスは，自分の作業領域を介してその腕木のヘッダ要素へ単一結合リストを作っている．一つのプロセスがある腕木に対してP命令を実行すると，マイクロプログラムにより腕木の値を一つ減らす．その結果の値が非負であれば，プロセスは実行を続ける．そうでなければその命令を中断し対応する腕木待ち行列の中に加える．腕木に対するV命令が一つのプロセスを解放するときは，腕木の待ち行列リストの中の最高の優先度を持ったプロセスを取り出す．最高の優先度のプロセスがいくつもあるときは，それらの中の一番古いものを取り出す．腕木の待ち行列の中で停止していないプロセスは**実行可能**（ready to run）か**実行中**（running）である．実行可能プロセスは“実行可能待ち行列”に結合し，腕木の待ち行列

と同じ様な順序でスケジュールする．こうすると，V命令によって解放されるプロセスは実行中のプロセスを先取りできるので，実行中のプロセスは実行可能待ち行列の中の最高優先度のものと少なくとも同じ優先度を常にもっている．

腕木はプロセスが相互に同期をとるとき，入出力装置と同期をとるとき，待ち行列を介してメッセージの交換をするときなどに用いる．最後の2つの応用については後節で述べる．

10.3.3 仮　想　記　憶

Venus 機械はマイクロ命令によって実現されるページ化されたセグメントの記憶を提供している．一つのページは256バイトの長さであり，一つのセグメントは256ページまで含むことができる．番地の参照は8個のレジスタの組の一つを用いて計算する．すなわち，セグメント・レジスタはセグメント名を持ち，それに対応する汎用レジスタはページ番号とその中の偏差とを持っている．各セグメント・レジスタはまた“拡張（extension）部分”を持っている．拡張部分とは汎用レジスタ内のページ番号をコアの番地（すなわち，そのページが存在するページ・フレーム）へ写像するレジスタである．記憶の1語にアクセスするためには，プログラムは汎用レジスタだけを参照する．マイクロプログラムは対応するセグメント・レジスタと拡張部分とを用いて，どのセグメントがそのページを含んでいるのかを決める．正しいセグメント名を適当なセグメント・レジスタにロードしてくるのは，プログラムの責任である．

セグメント・レジスタの拡張部分とページ・フォールトの機構は，マイクロプログラムの制御の下にある．汎用レジスタを介して1語をアクセスするとき，マイクロプログラムはセグメント・レジスタの拡張部分を調べる．もしもその拡張部分が汎用レジスタと同じページを参照していれば，そのページはコアの中にあり，その番地は拡張部分の中にある．その拡張部分が汎用レジスタの持つページを参照していないときは，マイクロプログラムはそれを捜さなければならない．第一に，現在，コアの中に存在するすべてのページのリストである**コア・ページ・テーブル**（core page table）を捜す．そのページがコアの中にあれば，セグメント・レジスタの拡張部分を調整してそのページを指すようにし，プログラムを継続することができる．そのページがコアの中になければ，

ソフトウェアの**ページ・フォールト処理システム**（page fault handler）を呼び出す．そして欲しいページのディスク上の位置を見つけ，コアのどこかにそれを格納し，コア・ページ・テーブルを更新し，マイクロ命令中の呼び出し地点まで戻る．ページ・フォールト処理システムはマイクロ命令のサブルーチンであり，したがって Venus 機械の一部分だと見なすべきことに注意しなければならない．

Venus におけるページの置き換えは，最長未使用（ＬＲＵ）規則によって行なう．どこかのセグメント・レジスタの拡張部分に付随しているすべてのページは，コアの中でロックしておかなければならない．さらに，コア・ページ・テーブルとかプロセスの作業領域とかページ・フォールト処理システムなどのように，プロセスやオペレーティング・システムのためにロックしておかなければならないページがいくつか存在する．他のページはすべて“年代順”のリストに結合し，置き換えの候補となっている．ページはどの拡張部分ももはや参照しなくなったときに，年代順のリストの最後につけ加える．したがって，年代順のリスト内のページは，それが最後にアクセスされた時からの長さに従って順序づけている．ページ・フォールト処理システムは年代順のリストから，先頭のページを取り出し，そのコア内のページ・フレームを新しいページの為に用いる．そして，セグメント・レジスタの拡張部分とコア・ページ・テーブルとを調整してから戻る．

プロセスや仮想記憶は，マイクロ命令により実現する．すなわち，Interdata 3は組み込まれたページ化のハードウェアとか多重プログラミングを提供してはくれない．それゆえ，記憶とプロセスの構成にはマイクロ命令を書いている設計者達の決定を取り入れてある．すなわち，本質的にはマイクロプログラム可能なハードウェアを用いれば，自分自身の機械をうち立てることができる．速度がもっと重要な機能のところ（たとえば，仮想記憶番地から物理的記憶番地を計算する場合）では，マイクロ命令を用いて実現できる．柔軟性が好ましいとき（たとえば，ページ・フォールト処理システムなど）は，その機能をソフトウェアにまで持っていくことができる．Venus の設計者達がハードウェアを設計するときは，彼等は実際には最低レベルの仮想機械に対する仕様を選んでいるのである．

Venusのマイクログラム可能な機械と，THEシステムの最低レベル２つとを比較してみよ．Venusシステムはファームウェアを用いて，独立な協力をして働くプロセスと仮想記憶とを実現しているのに対し，THEは同じ機能をソフトウェアで実現している．実際，これらの機能をハードウェアとして組み込むことができないという理由はない．THEのレベル１の**仮想**機械を**実**機械として実現すれば，そうしたことになる．

10.3.4　マイクロプログラムに対する拡張

Venusのマイクロプログラムの柔軟性をふやすような，ソフトウェアの追加が３つある．すなわち辞書，待ち行列およびポーラである．

10.3.4.1　辞　書

セグメントは15ビットの一意的な名前で識別するが，それはユーザ側からみると，特に便利なものというわけではない．そこで Venus のソフトウェアは外部のユーザが与えたセグメント名を内部の15ビットの名前に写像する辞書の集合を提供している．各外部名は一つの辞書とその辞書内の名前とを指定する．したがって，２レベルの外部の名前づけ機構を用いれば，ユーザは内部のセグメント名を追及せずにセグメントの共有を柔軟に行なうことができる．

10.3.4.2　待ち行列

腕木はプロセスの同期に対する機構を実現するために用いる．プロセスは待ち行列の集合を介して，メッセージを転送して通信を行なう．これらの待ち行列は相互排除を保証するために，腕木を用いてシステムが定義した手続きの集合だけを用いてアクセスする．

10.3.4.3　ポ　ー　ラ

Venusの中におけるプロセスの数は，コアの中に入れておくことのできるプロセス作業領域の数によって制限される．最大６個のユーザ・プロセスがあると仮定すると，システムに残されているのは10個のプロセスしかないが，これは充分ではない．したがって，**ポーラ**（poller）と呼ぶ単一の手続きを提供して，いくつかのプロセスが単一の仮想機械（すなわち作業領域）を共有することができるようにする．ポーラの仮想機械に影響を与える事象が生ずると，その事象に返事を出すまでその仮想機械を共有しているすべてのプロセスを走査

する．どのプロセスがポーラの仮想機械を共有するのかという選択は，システムの全体的な効率に基づいて行なう．

10.3.5 入出力システム

Venusは，すべての入出力機器とCPU，ディスク，システム・セグメントに対するデッドロックのない資源の管理を行なっている．Venusの資源管理に対するレベルを用いた方式の一番良い例は入出力システムである．このために，またそれはオペレーティング・システムの主要な要素であるということを考えて，入出力システムを詳細に記述する．

Venus入出力システムは仮想機械の系列（**図10-3**）として作ってある．レベル0は，マイクロプログラムした装置のチャネルである．チャネルはコマンドを受け付けて，コア内のデータのバッファを指定された装置へまたはそこから転送する．装置がテレタイプの端末，ライン・プリンタ，カード・リーダであれば，マイクロプログラムは一時に1バイトずつデータを動かして，実際に転送を行なう．テープとディスクの駆動装置に対するデータの転送は，ハードウェアのチャネルによって実行する．しかしながら，任意の装置における入出力の終了は，テープとディスクの駆動装置の場合も含めてマイクロプログラムしたチャネルによって受け付け，それから起動をかけたプロセスに対する腕木のV命令を実行することにより，入出力コマンドの終了を知らせる．

レベル1ではテープ駆動装置，カード・リーダ，プリンタ，各テレタイプに対して**管理者**（controller）**プロセス**を生成する（ディスクには仮想記憶のサブシステムが与えられている）．管理者はレベル0の仮想機械の上で実行し，ユーザに対し装置の特性を改善してくれる．レベル0のユーザは記憶の転送のためにコアのバッファを提供しなければならないのに対して，管理者は仮想記憶セグメントからバッファを受け取る．すなわち，管理者はセグメントのバッファからコアのバッファに対しデータを移す責任がある．さらに，転送の長さは一つのレコードでは固定しているが，レコードの大きさは（たとえば一枚のカードとか一行など）装置によって異なる．管理者は装置が空いていようといまいと，いつでもアクティブにすることができる．それは装置が使用可能となりしだい，入出力コマンドを実行する．要求した転送が完了したときには，コマン

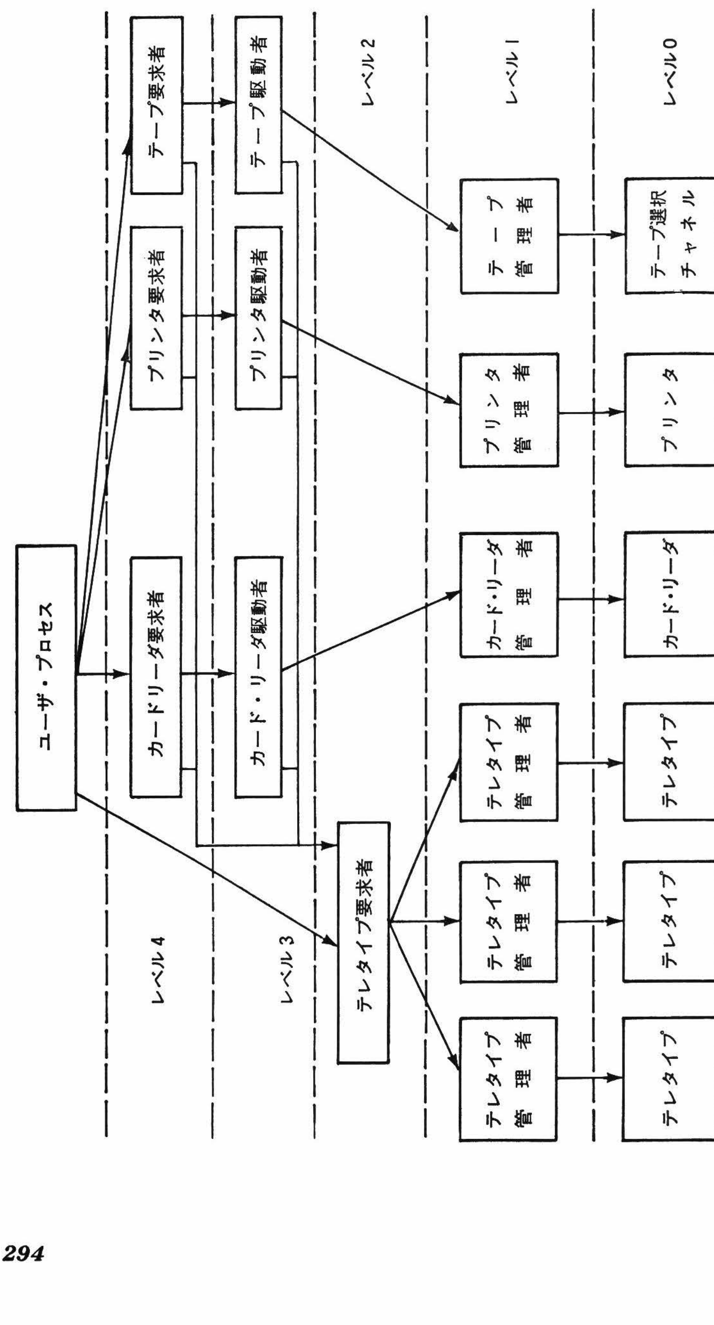


図 10-3　Venus 入出力システム

ド内に指定してある腕木を用いて要求したプロセスに伝える．

レベル2においては，**テレタイプ要求者**（teletype requester）と呼ぶ手続きを実現する．テレタイプ要求者はユーザおよびシステムのプロセスとテレタイプ管理者とのインターフェイスを提供する．それより高いレベルの入出力プロセスはテレタイプの端末と通信するためにテレタイプ要求者を用いることができる．

レベル3はカード・リーダ，ライン・プリンタおよび磁気テープ駆動装置に対する**駆動者**（driver）プロセスから成る．駆動者は入出力を行なうために管理者のコマンドを用いるが，一時に一つのセグメントを転送する仮想装置を提供する．それは入力バッファを単一のセグメントに併合し，また出力セグメントを管理者のために適当な大きさに分割する．カード・リーダの駆動者は，読取り機のホッパが空でない限り，カード・イメージからセグメントを作り出す．完全なセグメントを得るためには，同期をとることが必要である．印刷機の駆動者はセグメントを印刷するためのプロセスのコマンドによってアクティブにならなければならない．印刷が完了しても，そのユーザには知らされない．テープの駆動者はセグメントの読み込み／書き出しをするためのプロセスのコマンドによって，アクティブにならなければならない．呼び出したプロセスは読み込み／書き出しが完了するまで待たなければならない．

レベル4は各駆動者に対する**要求者**（requester）手続きから成り立つ．要求はセグメントを構成することおよびコマンド系列の同期を（たとえばテープの駆動者に対し）とることを助けることにより，駆動者とユーザ・プロセスとの間のインターフェイスを単純化している．

Venus の入出力システムは，Dijkstra の抽象化のレベルの方式を適用した良い例である．入出力システムの各レベルは，抽象化により不必要な細かさを取り除き，ユーザ・プロセスに対しもっと便利な仮想機械を提供してくれる．もとになる仮想機械すなわち Venus 機械自身は，腕木用命令とかページ化したセグメント記憶のような有益な道具を提供し，それにより入出力システムのすべてのレベルを実現しているのである．

10.4 他のシステム

文献にあらわれる他のシステムはほかにもたくさん存在する．この節では，そのようなシステムの2～3を興味をもつ読者のために紹介しておこう．

THE [Dijkstra, 1968 b] と RC 4000 [Brinch Hansen, 1970] オペレーティング・システムは小型の（かつあまり有名でない）機械に対して作られた高度に構造化したシステムである．両方のシステムとも新しい考えに基づいており，オペレーティング・システムをどのように構成すべきかという原型になるものとして多くの注目を浴びてきた．

MULTICS システム [Organick, 1972] は MIT で GE Honeywell の機械に対し実現した大規模で凝った計算機ユーティリティである．このシステムは柔軟な共有が実現できるような動的な環境を提供する複雑なセグメント化-ページ化番地付け機構を基本として作ってある．

いま述べたようなシステムは教育上の見地からは，良いものではあるけれども，実際の世界で一番多く使用しているオペレーティング・システムの代表ではない．Sayers [1971] と Hoare および Perrot [1972] は有名な機械に対する実用システムをたくさん紹介している．そのようなシステムを理解する第一段階として，読者は，通常接しなければならないようなシステムに関する情報を獲得するように努めるべきである．システムの外部特性を知ればその内部の機構を理解するのに役立つことが多い．

これらのシステムの大部分は構造的システム設計の考えが打ち出される前に実現されている．それらは理解しにくく，時には，使いにくい．その仕事においては SUE や Venus のような優雅なシステムを見つけることは稀であろう．しかしながら，実用的システムは多くの人々に対したくさんの有益な仕事をしてきたのである．

システムは優雅なものか実用的なものかどちらかしかなく，両者を備えることはできないと考えている人々がいる．われわれはこの哲学には賛成しない．それは自己敗北である．システムは実際の世界において，究極的な成功を収めることに反するような影響を与えずに，理解し易くきれいに構造化して設計することができる．

演 習 問 題

10.1 実在するオペレーティング・システムの事例研究についての準備をせよ．事例研究の例としては，次のものがある：

IBM OS/360 (MFT/MVT), DOS/360, 1800 Multiprogramming Executive, OS/VS, VM/370

DEC PDP-10 Time Sharing System, PDP-11 DOS System, PDP-11 RSX-11D, PDP-11 MUMPS, PDP-11 RSTS, PDP-8 Time Sharing System

CDC 6400/6500/6600 Scope 3 Operating System, CDC 3300/3500 Master Operating System

Univac 1108 Exec 8 Operating System

Honeywell 600 system, Honeywell 200 Mod 4 Operating System

MULTICS System

Compatible Time Sharing System (CTSS)

XDS Sigma 2 Real-Time Batch Monitor, Sigma 5/7 Batch Time Sharing Monitor

Burroughs B5500 system, B6500 Master Control Program

Hewlett Packard 3000 Operating System

ILLIAC IV Operating System

Michigan Terminal System

CAL 6000 Time Sharing System

10.2 ハードウェアの要素は値段が安くなり，能力は大きくなる傾向にある．従来，ソフトウェアで行なわれてきたオペレーティング・システムのいくつかの機能は，ハードウェアでとって替わることができる．オペレーティング・システムのどんな機能がハードウェアで実現するのに最も適していると考えられるか．オペレーティング・システム全体がハードウェア化されるべきではないというのは何か理由があるか．

10.3　銀行システムにおける計算機の使用について，概括をせよ．

10.4　American Airline の SABRE システム，Air Canada の予約システム，BOAC の予約システム（BOADICEA），United Airline の予約システム，IBM PARS など航空機の予約システムについての事例研究をせよ．

10.5　ミニコンピュータのオペレーティング・システムについて比較をしながら概括をせよ．

10.6　オペレーティング・システムで用いる術語は局所的な環境と構成とに依存して決まる．少なくとも3つの製造メーカか計算機装置の主要なユーザのマニュアルに出てくる術語の辞書を作成せよ．一般によく使われる概念，たとえば，プロセスのすべての異った名称を含ませることを保証せよ．

補遺 I

データ構造

データ構造 (data structure) は相互に構造的な関係がある一つ以上のデータ要素の集合である．データ構造を扱う際の２つの基本的な問題は，**生成** (creation)と**保守** (maintenance) である．後者の処理は構造の中に要素を**挿入**(insertion)すること，構造の中から要素を**削除** (delete) をすること，構造の中で要素を**見つける** (locate) ことなどの操作を含む．各々の応用で，前述の操作の一つを強調するので，データ構造は非常に多様性に富んでいる．そこで，その操作を空間および計算の必要量から見て，非常に効率よく行なう試みがなされる．

複雑なデータ構造を用いることは，オペレーティング・システムのあちこちで見られる．この章では，構造化した情報の格納および検索について，いくつかの基本的概念を概説しよう．これは入門的な提示を行なう積りではなく，むしろ本書のバックグラウンドとして必要な題材を概括する積りである．もの足りない読者は，たとえば Knuth [1968] の著書のようなデータ構造に対する良い教科書を読むことを奨める．

I.1 術語の定義

情報の論理的な単位を**要素** (element) と定義すると，**リスト** (list) は線形に並んだ要素の系列 $[e(1), e(2), \cdots e(n)]$ である．すなわち，２つの要素の組 $e(k)$ と $e(k+1)$ とは，$e(k)$ が $e(k+1)$ の前にあり，$e(k+1)$ は $e(k)$ の後にあるという関係にある．リストは順序づけられた情報の表が必要なところで形成できる（たとえば，使用可能な記憶のブロックのリスト，アクティブなジ

ョブのリスト，ユーザ・ファイルのリストなど）.

リストへの要素のつけ加え方，リストからの要素の取り除き方に制限を課すことにより新しい構造を定義する．たとえば，**スタック**（stack）とは挿入と削除とを片側だけで行なうリストと定義する．**キュー**（queue）とは挿入と削除とを両側で行なうリストである．

I.2 リストの順次的割り当て

リストの項目は記憶の連続的場所に割り当てることができる．MEMORY (y) を y と名づけた項目の物理的記憶の番地としよう．あるベース・アドレス（すなわち，MEMORY ($x(0)$) が基礎番地である）に第一要素を格納しているようなリスト x を考えよう．こうすると，各項目が L という固定長の大きさであったとき，順次的割り当ては次の式で示される：

$$\text{MEMORY}\ (x(i+1))=\text{MEMORY}\ (x(i))+L,\quad i\geq 1$$

同一の名前の領域内に順次的に割り当てた4つのリスト (A, B, C, D) が，記憶内に存在すると仮定しよう．**図 I-1** では，C と D の右側には未使用の場所が充分あるけれども，領域 B はリスト C が使用している領域をのり越えてしまう．このような状態で問題を解決するためには，よく使用する2つの方策が適用できる．

第一の方法は，**詰め**（compaction）と呼ぶが，小さいリストを再配置して，大きなリストに対する空きを作ろうとすることである．詰めを適用すれば，B が使用できる空間は，**図 I-2** に示すように拡張できる．C と D とを大きい番地に移すことにより，B に対して空きの空間が追加された．

第二の方法は，**間接法**（indirection）と呼ぶが，これはリンクを使用する方法である．**リンク**（link）とは記憶の別の語を指す番地を含んでいる記憶の要

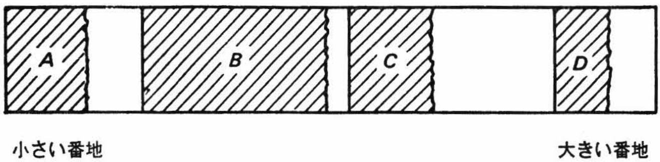


図 I-1 記憶マップ I（斜線を施した領域は，使用している領域を示す）.

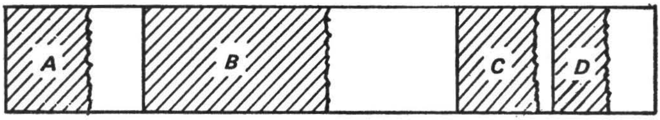


図 I-2 記憶マップ II.

素である．空いた空間を指すために混んでいる領域の後にリンクを置けば，大きなリストは記憶を再配置しなくても追加の領域を拡大することができる．**図 I-3** では，図 I-1 の領域 B を間接法を用いて拡張している．記憶の隙間を埋めるために間接法を用いると，順次的割り当てはリスト内の連続領域の中だけで適用できる．これは詰めの場合とは，対照的である．この場合，順次的割り当てがリスト全体の中で維持される．

I.3 結合リスト

（単一の）結合リストの表現においては，リストの各要素は2つの項目を持っている．すなわち，その要素に与えられたデータ（DATA と呼ぶ）とリスト内の次の要素を指すリンク（POINTER と呼ぶ）とである．**図 I-4** には4つの要素から成る結合リストを示してある．**リスト・ヘッド** (list head) と呼ぶ特別な要素がリストの最初の要素を指し示す．最後の要素はリストの終りを示すために，“空”のポインタを持っている．実際の記憶位置はリストの論理構造とは無関係なので，リンクは要素の番地よりもむしろ矢印で示されている．

リストは**二重結合** (doubly linked) にすることもできる．要素 x は2つのポインタを持つとき**二重結合**であるという．一つは x の左の要素を指し，もう一つは x の右の要素を指す．4つの要素から成る二重結合リストを**図 I-5** に示す．

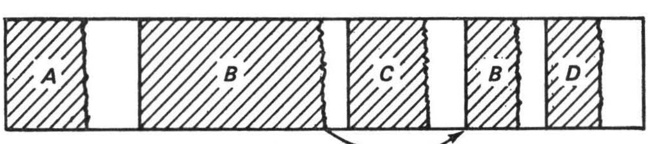


図 I-3 記憶マップ III.（ポインタはリストBの続きを示す）.

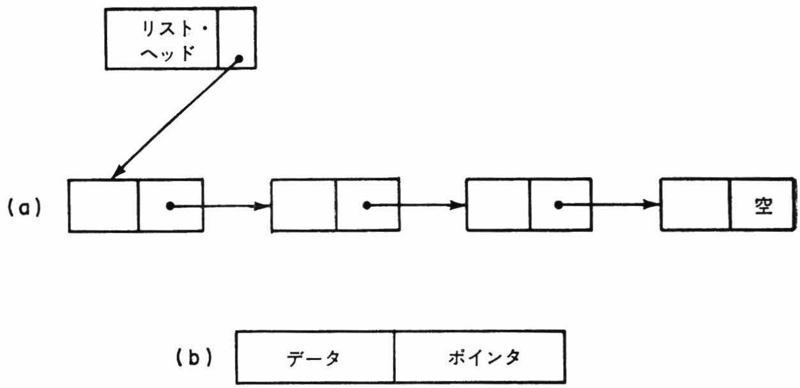


図 I-4 (a) 各要素が (b) の形式を持つ単一結合リスト

二重結合リストは単一結合の片われに対し50%余計な記憶(余分なポインタ)が必要ではあるけれども，リストへの挿入あるいはリストからの削除が単純化される．要素へのポインタだけを与えられた場合，その左へのポインタがわかっているので二重結合リストから要素を削除するのは容易にできる．単一結合リストにおいては，削除する要素を指すために用いていたポインタを修正するために，リストの先頭からたどらなければならない．

循環リスト (circular list) では最後の要素が，先頭の要素を指し示しているリストであり，要素がループを構成している．要素内にリンクのフィールドを

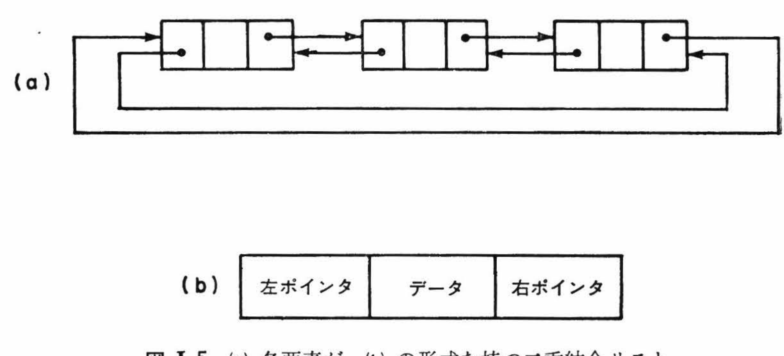


図 I-5 (a) 各要素が，(b) の形式を持つ二重結合リスト

ふやせば，もっと複雑なデータ構造が得られる．これらの**多重結合リスト**（multiply linked list）は任意のグラフ構造（たとえばサイクル，木，ネットワーク）を構成することができる．

I.4 結合リストの管理

結合リストのどんな管理の方式においても，未使用の要素を記憶から取ってくるという操作が必要になる．よく用いられる方式は，未使用の要素を単一結合リストにした**自由リスト**（free list）を維持する方法であり，そのリストはヘッダ要素により指し示されている．自由リストに要素を追加するのには基本的に2つの方式がある．

I.4.1 ユーザで定義する管理

この方式では，要素が必要なくなったときに自由リストに戻すのはユーザの責任で行なう．要素xを自由リストに戻す一つの方式は，xの POINTER を自由リストの先頭の要素を指すように設定し，自由リストのヘッダ要素がxを指すように調整する方法である．こうするとxは新しい自由リストの先頭の要素になる．プログラムが実行を開始する前に記憶を全て結合することにより，自由リストを初期設定できる．すなわち，自由リスト・ヘッダは記憶の先頭の要素を指し示し，最後の要素は“空”を指す．しかしながら，記憶がかなり大きければ，この手続きはかなり時間がかかる．別の方法として記憶に限界を設けて，それを自由要素として用いる．最初のうちは上限に達するまで順次的に要素を割り当ててゆく．使用した要素は，最初空であるような自由リストに戻してゆく．最初に決めた制限を越えたときは，自由リストを介して管理技法を陽に開始する．この方法は次の方式でも応用される．

I.4.2 システムによる管理

削除した要素を扱うユーティリティ・プログラムを書けば，プログラマは使用していない要素を自由リストに陽に戻すという義務から解放される．2つの方式が可能となる．

一つは**使用カウント**（use count）または**参照カウント**（reference count）で

ある．各要素xに新しいフィールドをつけ加え，そこにxを参照している他の要素の数（すなわち，他の要素のリンクがいくつxを指しているのか）を格納しておく．要素をシステムに追加したりシステムから取り除いたりすると，参照カウントが頻繁に更新される．参照カウントが0になると，その要素をプログラムはアクセスできなくなり，再割り当てのために使用可能となる．したがって，システムは要素を自由リストに安全に戻すことができる．

この方法には，いくつかの問題がある．第一に，循環リストに組み込まれている要素は自由な記憶に戻ることは決してない．プログラムがアクセスできるリストはどれも循環リストを指していなくても，循環リスト内の0でない参照カウントのために，それは解放されなくなる．これは特に循環リストがもっと複雑な構造の中に埋め込まれているときに解決するのが難しい．第二に，参照カウントを記録する各要素の中の余分なフィールドは，使用可能な記憶のかなりの部分を“無駄”にしてしまう．この問題は各リストのヘッダ要素内にだけカウントを記録して，必要な記憶の量を幾分か減少させることにより解決できることもある．しかし大部分のリストが短いものであると，無駄はやはり重要である．第三の問題は，プログラムが誤動作のために死んでしまうときに生じてくる．プログラムはもはや項目にアクセスできないのに，参照カウントは減少していない．それゆえ，そのリストは再割り当てされることは決してない．しかし，時にはこの性質が望ましいこともある．死んだプログラムが再開始したとき，参照カウントを用いてデータ構造を再構築することをうまく試みることができる．

もう一つの方式は**屑拾い**（garbage collection）と呼ぶ．この方法では，ユーザのプログラムは自由記憶がこれ以上使用できないというところまで実行を続けるが，その時点で，“屑拾い”のプログラムを呼び出して，捨てた要素をすべて拾い上げる．屑拾いのアルゴリズムでは各要素内の“マーク・ビット”を利用している．そのルーチンは，すべての要素のマーク・ビットを0に設定することからはじめる．次にユーザ・プログラムが直接アクセスできる固定した記憶の要素（リスト・ヘッドであることが多い）から開始して，この基本要素からポインタをたどることにより到達できるすべての要素のマーク・ビットをひっくり返す．マークされない要素すべてはユーザからはアクセスできないも

のと考えられ，自由記憶リストにつなぐことができる．この方法については，2つの重要な問題点がある．第一は，記憶がほとんど一杯になってしまうと，屑集めのプログラムはそれ自身のための作業領域が少なくなってしまい，したがって，ゆっくりとしか動かない．さらに大部分の要素がまだ使用されているならば，マークしなければならないビットはたくさんある．第二に，屑集めがリストの処理中に呼ばれたとき，ユーザのプログラムには一時的にはつながっていないので，実際のリストの部分を屑として扱い，事故が生ずるかもしれない．アクティブなリストの部分と実際の屑とを見分けることは容易ではないことが多い．

屑拾いや参照カウントを用いれば，アクティブなリストの要素の間に自由記憶の小さいブロックがばらまかれる状態になり，記憶は断片化する．詰めはマーク・ビット（または使用カウント）を設定した後で適用すれば，現在アクティブであるようなすべてのリストの要素は，記憶の先頭の連続した場所に移される．この過程を経ると，すべての自由な要素は記憶の終りの連続した場所に見つけることができるので，自由リストを作る必要性がなくなる．大部分の要素がアクティブでなければ，自由リストをつなぐ必要がないので，この方法によりかなりの時間と空間とが節約になる．他の技法と同様に，詰めがどうしようもなく能率の悪くなるような病的な場合も作り出すことはできる．

I.5 スタック

LIFO（後入先出）**リスト**は通常**スタック**というが，これはすべての挿入と削除とが片側だけで行なわれるリストである．スタックに加えたり取り除いたりする操作は，それぞれ **PUSH** と **POP** と呼ぶ．順次的割り当て方式では，スタックはベース・アドレスとスタックの先頭に対するポインタにより実現することができる．PUSH と POP はまたヘッダ要素を持つ単一結合リストにおいても，実現することができる．スタックに対し要素 x を PUSH するためには，x の POINTER にはスタックの先頭の要素の番地（それはスタック・ヘッダに格納されている）を代入する．次に，スタック・ヘッダには x の番地を代入する．スタックから要素を POP するためには，スタック・ヘッダの内容を自由変数のどれかに代入し，ヘッダは前に先頭であった要素の POINTER が

指していたところを指すように調整する.

恐らく最も広くスタックが使用されるのは，言語の構文解析のアルゴリズムとグラフの走査の場合である．スタックはオペレーティング・システムよりもコンパイラにおいて頻繁にみられるけれども，ここでは完全性を期すために紹介しておいた.

I.6 キュー

FIFO（先入先出）**リスト**または**キュー**は，すべての挿入は片側で，またすべての削除はもう一方の側で行なうようなリストである．キューに対する **INSERT** および **DELETE** 操作はスタックに対する PUSH と POP とに相当するが，これは2つのポインタ——一つはリストの先頭に対するもの（挿入のため），もう一つはリストの終りに対するもの（削除のため）——が必要となる．キューにおいては（あるいはスタックにおいても），アンダーフロー（すなわち，空きのリストから削除をすること），および，オーバーフロー（すなわちいっぱいのリストに挿入すること）は誤りの条件として扱わなければならない.

FIFO キューはオペレーティング・システムにおいては，いろいろの応用ができる．余計な情報が与えられていなければ，ある資源を待っている**ユーザは**，一般に先来先処理（FIFO）を基礎にして処理する.

I.7 デキュー

デキュー（dequeue）（二重入口キュー）においては，挿入と削除はそれぞれリストのどちら側において行なってもよい．FIFO の場合と同様に，デキューの各側に対応する2つのポインタが必要となる．順次的割り当ての場合には，リストは両方向にふえてゆくので，記憶に対して上限と下限とを見張る以外には LIFO や FIFO と比べて，新しくでてくる難しい問題はない．しかし，結合リストとの場合は非常に複雑になる．スタックやキューと異なり前の要素を捜さないで，両側から要素を削除することはできないので，単一結合リストでは不充分である．解決の方法はリストを二重結合にすることであるが，その結果，挿入と削除のアルゴリズムが複雑となり時間がかかるようになる．デキューは，また両方の端に対する内部ポインタを持った循環リストとしても作るこ

とができる.

I.8 表

表（table）は，挿入と削除とがどの点においても許されているデータ構造である．表を実現するために一般的に用いられる機構が，いくつか存在する．**ハッシュ・コーディング**（hash coding）技法においては，表内の要素の内容や名前をその要素を格納する番地を計算するために使用する．たとえば，要素が一意的な名前を持っていれば，各名前の内部（すなちビット）表現を表の限界内にある番地を得るために，単純な算術変換として用いることができる．理想的には，ハッシュ・コーディング関数は異った要素に対し異った番地を生成してくれる．これが成り立たない場合，同一の番地にばらまかれた要素を結合しておく．たとえば，ハッシュ・コーディング関数が名前AとTとを一つの番地Kに写像したと仮定する．その要素のうち，一つだけをKに格納できるが，たとえばAを格納したとしよう．要素Pはどこか別の番地K'に格納し，Kの要素はK'に結合する．第三の要素が番地Kにばらまかれたならば，それを新しい番地K''に格納し，K'の要素をK''に結合する．プログラムがハッシュ・コーディング関数を用いて表にアクセスするときは，要素の名前がばらまかれる場所に実際望んでいる要素が格納されているかどうか，チェックしなければならない．その場所に格納されていなければ，プログラムはその場所から開始して，結合リストをたどり望みの要素を捜し出さなければならない．

表の番地づけを行なう第二の方法は，名前を**ディレクトリ**（directory）に格納し，要素の番地も一緒に格納する方法である．要素にアクセスするときは，要素がどこに格納されているかを決定するために**表引き**（table lookup）を行なう．表そのものを捜すのではなく，分類したディレクトリを捜す方が時間が少なくて済むという仮定をしている．ディレクトリの探索をハードウエアで実現する（通常，並列探索を用いる）ときには，**連想記憶**（associative memory）を持つことになる．すなわち，ハードウエアは高速に探索できる特殊なハードウェア・レジスタを用いて，要素の名前と記憶の場所とを対応づける．ハッシュ・コーディングとディレクトリとは，表を直接探索しないで要素を捜し出そうという点で似ている．

補遺 II

計　算　構　造

II.1　はじめに

いくつかのプロセスを，並列的に実行しているようなシステムにおいては，順次的システムに存在していなかったような問題がいくつかあらわれてくる．オペレーティング・システムの分野における理論的研究の大部分は，並列システムの性質を解析するためのモデル（**プログラム図式**（program schemata）と呼ぶことがある）を開発することに集中してきた．実際，文献には25個以上の異ったモデルが発表されている［Bernstein, 1973］．この補遺では，これらのモデルの中から3つを選び，代表的な例として紹介している．すなわち，ペトリ・ネット，計算図式および再使用型資源グラフである．

II.2　ペトリ・ネット

ペトリ・ネットは，一般的なシステムにおいて操作の並列性を記述する道具として導入された［Holt および Commoner, 1970］．これは計算機システムにおけるデットロックに関するいくつかの興味のある問題を解決するために使用することができた．さらに，ペトリ・ネットは並列計算に対する多くの異ったモデルにおいて，複雑な形で埋め込まれていることがわかった．したがって，これは図式の議論をするのによい手がかりとなる．

ペトリ・ネット（Petri net）は，2つの型の節点**場所**（place）と**遷移**（transition）とを持ち，各辺は場所から遷移あるいは遷移から場所をつないでいるラベル付きの有向グラフとして定義する．場所は円で遷移は横棒でまた辺は有向枝で表わす．ペトリ・ネットの**マーク付け**（marking）Mは，場所のラベル

から非負整数への関数である．マーク付けは，各場所にマーク付け関数に応じていくつかの**石**(token) を割り当てることにより表わす。Nをペトリ・ネットとし，PをN中の場所の集合，$n(P)$ をPにおける場所の数とする．Nの中の場所は名前の集合 $\{1, 2, 3, \cdots, n(P)\}$ から選んだ一意的なラベルを与えてある．マーク付けMは $n(P)$ 個の要素から成るベクトルと考えるが，そこではベクトルの i 番目の要素は，i 番目の場所の石の数をあらわす．これらの概念を**図II-1**に示す．

ネットのマーク付けは遷移を**ファイアすること** (firing) によって変化する．遷移 t は t へ辺が向っているようなすべての場所 x に対して少なくとも一つ石が存在するとき，**ファイア可能** (firable) であるという．遷移 t のファイア(t はファイア可能とする）により次のような変換が行なわれる．すなわち，場所 x から遷移 t へ向かう各辺に対し，x 上の石の数を１つ減らす．t から場所 y へ向かう各辺に対し，y 上の石の数を１つ増やす．**図II-1**をもう一度見てみると，遷移 a と b とはファイア可能であるが，遷移 c と d とはそうではない．遷移 a がファイアすると，マーク付けは $M'=[1,1,0]$ になる．今度は遷移 a, b, c が

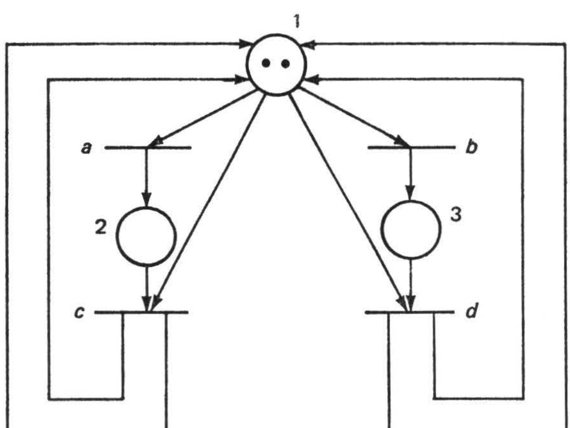


図 II-1　停止する可能性のあるペトリ・ネット（このペトリ・ネットに対するマーク付けMは $M=[2,0,0]$; $P=\{1,2,3\}$; $n(P)=9$; a, b, c, d は遷移）

ファイア可能となる．c がファイアすると新しいマーク付けは $M''=[2,0,0]$ となる．

ペトリ・ネットにおいては，遷移の並列的な実行に対する唯一の制限は場所の中の石の数で決まる．したがって，**図II-1**における最初のマーク付けにおいては，遷移 a と b とは同時にファイアすることができる．2つの遷移が同一時にファイア可能であるということが必ずしも並列的にファイアできるということを意味してはいない．**図II-1**のマーク付けが $M=[1,0,0]$ であった場合，a と b 両者ともにファイア可能である．しかし，同時にファイアすることはできない．何故ならば，ファイアを行なうためには a と b とはそれぞれ場所1から石を1つ（すなわち全体で石2つ）取り去る必要があるからである．

意味論的には，場所はシステムにおける**条件**（condition）を，また遷移は**事象**（event）をあらわす．遷移 t の入力場所（すなわち t に向かっている辺を持つ場所）は，事象 t の生起に影響を与える条件に対応する．t の出力場所はその事象の生起によって，システムに与える効果に対応する．石は場所によってあらわされる条件が成立するとき，その場所に存在する．ファイアする操作は一つの事象を左右するすべての条件が成立したときにその事象が生起し，その出力条件が成立するという意味を持つ．

初期のマーク付けが $M(0)$ であるようなペトリ・ネット N を考えよう．N の**実行系列**（execution sequence）とはマーク付けの系列 $M(0), M(1), M(2), \cdots$ である．ただし，$M(i+1)$ は遷移をファイアすることにより $M(i)$ から得られるものとする．一般に，与えられたペトリ・ネットと初期のマーク付けに対し，可能な実行系列が多数存在する．マーク付けは $M(0)$ から開始してすべての可能な実行系列が無限であるとき**生きている**（live）という．マーク付けがどの遷移もファイア可能でないようなところまで達したとき，ペトリ・ネットは**停止する**（hang up）（すなわちデッドロックする）という．システムが生きているいくつかの初期のマーク付けを持つペトリ・ネットとしてモデル化できれば，どの事象も生起しないようなマーク付けに達することは不可能であるので，このシステムは全体として決して停止しない．

図II-1において，初期のマーク付けを $M(0)=[2,0,0]$ と仮定しよう．a と b が各々ファイアすれば，マーク付け $M(2)=[0,1,1]$ に到達する．ペトリ・

ネットは，そこで死んでしまう．どの遷移もファイアできない．それゆえ，マーク付け $M(0)$ は生きてはいない．実際，このペトリ・ネットに対しては生きているマーク付けは存在しない．どんなに多くの石を場所1に置いても，遷移 a と b とをファイアし続けすべての石が場所2と3とにくるようにすることが常に可能である．場所1が空になれば，すぐにこのペトリ・ネットは“死んで”しまう．

2つの遷移は，お互に相互排除的（すなわち同時刻に両方ともファイアすることはできない）ならば，**衝突**（conflict）の状態にあるという。衝突が生ずるこまかい条件は形式化されていないけれども，衝突の概念は**図II-1**にあらわれている停止状態の型と密接な関係にある（すなわち，遷移 c と d はマーク付け $M(0)$ の下では衝突している）ことは明白である．

マーク付きグラフ（marked graph）とは，すべての場所においてただ一つの辺だけが出ており，また入ってきているようなペトリ・ネットである．マーク付きグラフは，ペトリ・ネットに比べて幾分一般的ではなく，したがって解析し易い。停止と有限性とに関連したマーク付きグラフに関する定理がたくさん証明されている．特に強連結のマーク付きグラフに対しては死んだ状態（すなわち，ファイアが決して起らない）が存在する必要十分条件は，石を一つも持っていない有向サイクルが存在することである．遷移がプロセスに対応するな

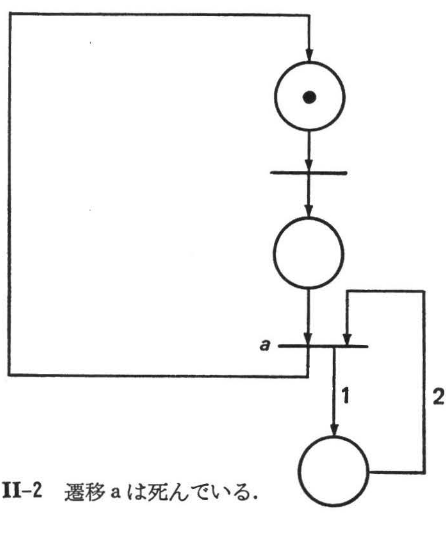


図 II-2 遷移 a は死んでいる．

らば，この定理はマーク付きグラフによってモデル化されるシステムに対して意味を持っている．一般的なペトリ・ネットに対して，比較されるような命題は見つかっていない．**図 II-2** のマーク付きグラフにおいては，辺1と2は石を持たない有向サイクルを構成するので，遷移 a は死んでいる．

ペトリ・ネットNにおけるマーク付けMは，遷移をファイアしてMから得られるすべてのマーク付けの中で，一つ以上の石を持つ可能性のある場所がないとき**安全**（safe）であるという．マーク付きグラフにおいて，生きていること(liveness）と安全性との関係に関する結果はたくさんある．

ペトリ・ネットは，すべての遷移が高々一つの入力と出力だけしか持たないとき，**遷移図**（transition diagram）であるという．マーク付きグラフと遷移図に対してはきれいな定理がたくさん得られているけれども，オペレーティング・システムにおける関係は，通常一般的なペトリ・ネットの完全な記述能力を

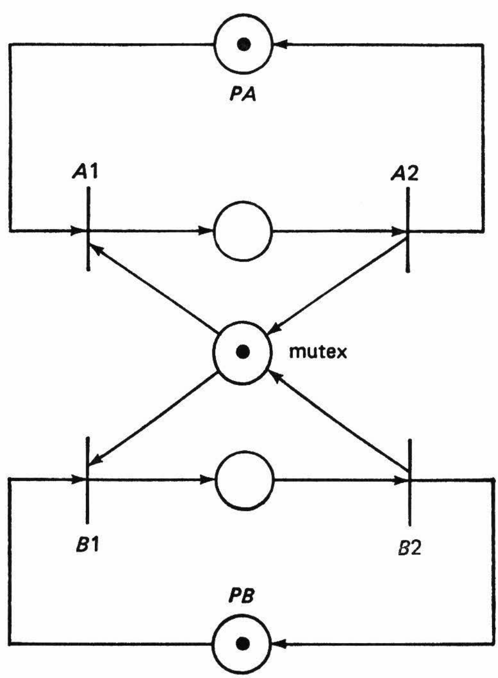


図 II-3 ペトリ・ネットを用いた相互排除

必要としている．

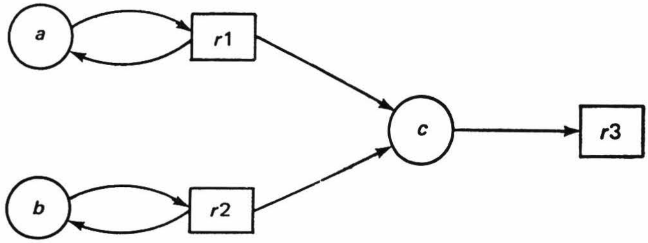
たとえば，2つのプロセスAとBの間の単純な相互排除関係を考えてみよう（図II-3）．遷移 $A1(B1)$ はプロセス $A(B)$ がその危険部分に入ろうと思うときファイアする．遷移 $A2(B2)$ はプロセス $A(B)$ がその危険部分を去るときにファイアする．$A1$と$B1$とが相互排除的であるためには，入力場所（すなわち"mutex"）を占有しなければならない．こうすると mutex は二つの出力を持っているので，このペトリ・ネットはマーク付きグラフではない．また$A1$と$B1$とは，それ自身，準備された（すなわちPAとPBとが石を持っている）ときおよび mutex が使用可能な（すなわち mutex が石を持っている）ときにのみファイアをすることを希望するので，$A1$と$B1$とはそれぞれ2つの入力が必要となる．したがって，このペトリ・ネットは遷移図ではない．2つのプロセス間の単純な関係でもこのようにモデル化すると，一般的なペトリ・ネットになってしまうことを考えると，もっと一般性の少ないモデルに関する定理を適用するという気持は特に強くなるわけではない．

Ⅱ.3 計 算 図 式

ペトリ・ネットはシステム内の事象の相互作用を記述するのに充分であるけれども，計算をするという見地からすると，それは完全に一般的な形をしているわけではない．たとえば，"否定"操作（すなわち遷移がファイアする必要十分条件は場所が空のときである）は，ペトリ・ネットによってモデル化することはできない．この節では，計算的には完全な図式の例を概説する．**計算図式** (computational schema) とは**レジスタ** (register) の集合（すなわち記憶場所）に働く**演算子** (operator) の集合（すなわち，プロセス）から成る非同期システムを図形的な形式で表現したものである．各図式は2つのグラフ，すなわち**データ流れ図** (data flow graph) と**制御図** (control graph) によって完全に定義される．

II.3.1 諸 定 義

データ流れ図は各演算子の**定義域** (domain) と**値域** (range) のレジスタを指定する．演算子zからレジスタrへ向かう有向枝はrがzの値域であること



演算子	定義域	値 域
a	$\{r_1\}$	$\{r_1\}$
b	$\{r_2\}$	$\{r_2\}$
c	$\{r_1, r_2\}$	$\{r_3\}$

図 II-4 データ流れ図

を示す．r から z へ向かう有向枝は r が z の定義域に属することを示す．**図II-4**には計算図式のデータ流れ図を示してある．演算子とレジスタとは，円と四角でそれぞれ表現している．

制御図は演算を実行する順序を指定する．すべての演算子（円であらわす）はいくつかの**制御カウンタ**（control counter）（四角であらわす）につながっているが，制御カウンタはそれぞれ非負整数の内容を持っている．四角の中に書いてある数はそのカウンタの**初期値**である．計算図式の制御図の例を**図II-5**に示す．

演算子節点に向かっているカウンタ（すなわち**入力**カウンタ）がすべて0より大きい値を持っていれば，その演算子は**定義されている**（defined）という．任意の時点において，演算子が定義されていればそれは実行してもよい．その結果データ流れ図に従ってレジスタを更新し，以下のようにして制御図を修正する．すなわち，その演算子に対するすべての入力カウンタを1だけ減少し，すべての出力カウンタをある非負整数の値だけ増加する．出力カウンタの増加が一定値であれば，**図II-5**の演算子 a と c のように辺の上に書いておく．順序がデータによって左右することを許すために，出力カウンタの更新は**図II-5**の

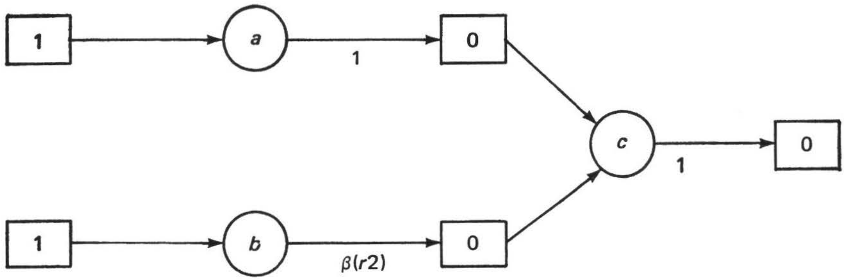


図 II-5 制 御 図

演算子 b のように演算子が参照するレジスタの値の関数としてもよい．

制御図とペトリ・ネットの相似性に注意しよう．演算子とカウンタとを遷移と場所と考えるならば，2つの表現の重大な差異はデータに依存したカウンタの増加があることである．

通常，データ流れ図の演算子は任意の関数として表現し，また（カウンタの値に基づいた）制御上の決定は任意の述語として表現する．このような場合，図式は**解釈していない**（uninterpreted）という．図式の**解釈**（interpretation）とは，実際の関数と述語とを関数と述語の記号に割り当てることである．解釈は，一般にレジスタの値に対する定義域を含んでいる．**図II-4**および**図II-5**で定義される図式は解釈していない．この図式に対する解釈は，次のようなものになるであろう：

1. $r1, r2, r3$ に対する値の定義域．
2. 演算子 a, b, c に対する関数の仕様．
3. 制御図における $\beta(\ )$ に対する整数値の関数．

図式の**実行系列**とは，演算子の名前の（無限でもよい）系列 $X1, X2, \cdots, Xn, \cdots$ である．ただし各演算子 Xi は，その系列の中で前に存在する演算子を実行した結果得られたカウンタの値を持って定義されている（すなわち，すべての入力カウンタは1以上）．実行系列 $X1, \cdots, Xn$ において，$X1, \cdots, Xn$ の実行が修了した後では，どの演算子も定義されていないとき，それは**完全**（complete）

表 II-1 計算図式の履歴配列†

レジスタ	値	
r_1	x_1	$a(x_1)$
r_2	x_2	$b(x_1)$
r_3	x_3	$c(a(x_1), b(x_2))$

† 図 II-4および II-5のグラフで表現される図式を考えよう．完全な実行系列は，a, b, c および b, a, c だけである．この履歴配列は，これらの系列の両者に対応する．

であるという．各々の実行系列は各レジスタに対するその内容の系列を持っているような**履歴配列**（history array）を定義する．**表 II-1** は実行系列と履歴配列との関係を示している．

II.3.2 決定性，関数性，デッドロック

演算子に対しては特別なタイミングを付与していないので，演算子が開始し終了する順序は常に予測できるわけではない．任意の正しい実行系列は，事象の可能な系列である．一般に，図式を実行するときはいつでも“同じことをする”ことを期待している．しかしながら，計算の結果が実行系列に左右されることがある．図式がすべての解釈のもとで，すべての正しい実行系列に対し同一の結果を生み出すならば，その図式は**決定性的**（determinate）であるという．

図 II-6 に示した計算図式を考えよう．演算子 a がレジスタ$r1$に 0 を格納したと仮定しよう．演算子 b は$r1$を読み，それを$r2$に格納する．あきらかに，$r2$

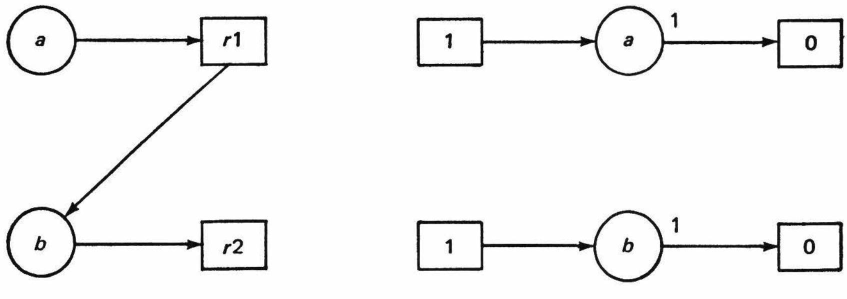


図 II-6 非決定性的計算図式

の結果は b を a よりも前に実行するか，後で実行するかによって異ってくる．しかし，制御図は両者の実行系列を許している（すなわち a,b または b,a）．したがって，この図式は非決定性的である．

2つの演算子は，レジスタにおいてどちらか一方が r を変更し，もう一方が r を変更したり参照したりするとき**衝突**するという．どこかのレジスタで衝突する2つの演算子が同時に定義されるとき，図式には**競争条件**(race condition)が存在するといい，その図式は非決定性的である．**図II-6**においては，演算子 a と b がレジスタ $r1$ で衝突している．両方とも最初から定義されているので，競争条件が存在する．

図式はすべての実行系列に対し，すべてのレジスタの値が同一な系列を示すとき，すなわちその図式に対するすべての履歴配列が同一のとき**完全に関数的**(completely functional）であるという．競争条件が存在するときは，図式はす

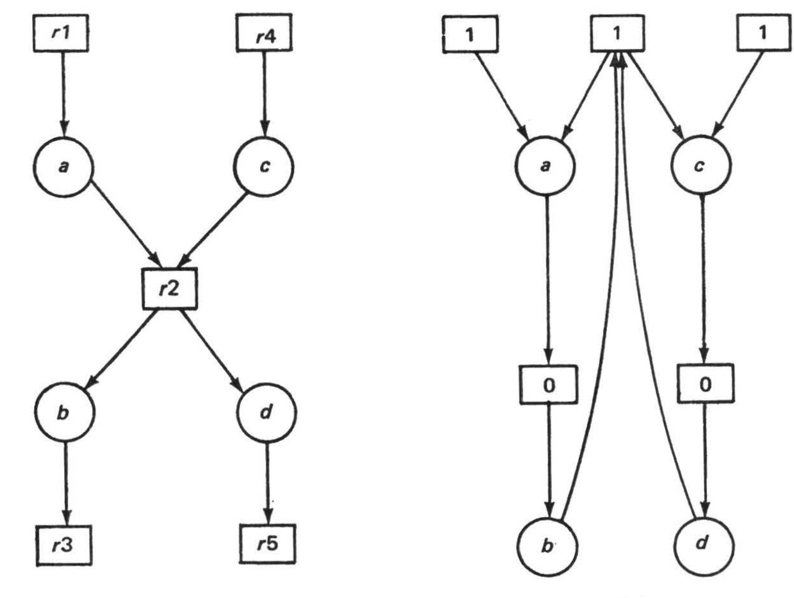


図 II-7 出力関数的なものの例．(a)データ流れ図；(b)制御図（レジスタ $r3$ と $r5$ とが，出力レジスタであれば，この図式は，出力関数的である．$r2$ が二つの履歴（すなわち，［$a(r1), c(r4)$］または［$c(r1), a(r4)$］)を持つ可能性があるので，完全関数的ではない)．

べての解釈に対して，完全に関数的になることはないであろう．

図式が完全に関数的になることを要求するかわりに，レジスタの部分集合を**出力レジスタ**（output register）と定義し，出力レジスタに対するすべての履歴配列がすべての実行系列に対して等価なとき，その図式は**出力関数的**（output functional）であるという．**図 II-7** においては，$r3$と$r5$だけが出力レジスタであれば，この図式は出力関数的である．$r2$ が2つの履歴（すなわち [$a(r1)$, $c(r4)$] または [$c(r4), a(r1)$] を持つ可能性があるので，これは完全関数的ではない．

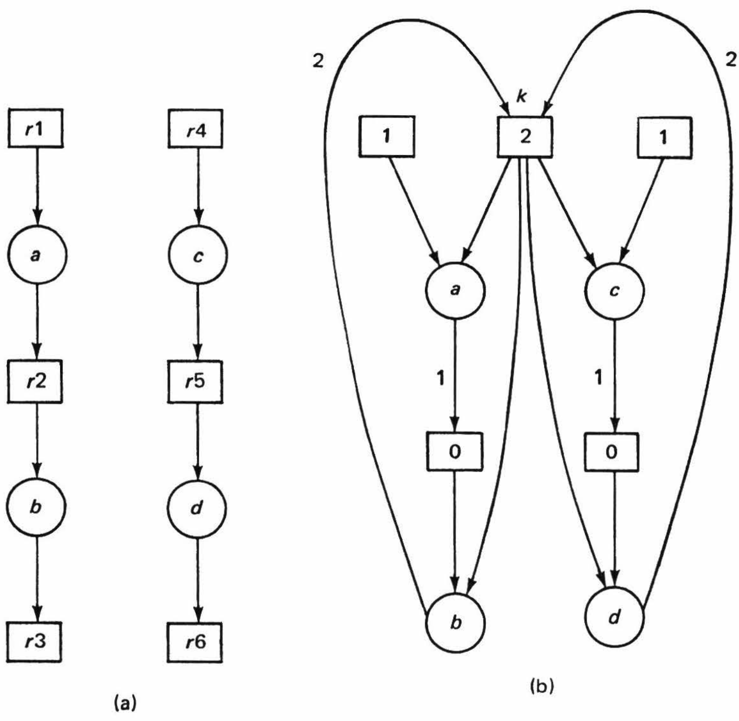


図 II-8 停止する図式の例．(a) データ流れ図；(b) 制御図（kとラベルしたカウンタは，各演算子で必要とする資源をあらわすが，bまたはdが完了する迄，解放されない．実行系列 a, b, c, d と，c, d, a, b は，$r3$ $r6$ に意図した結果が入るように計算する．しかし，aが最初に実行され，cがそれに続くと，bもdも，進まなくなる．そして，実行は，正しく終了しない）．

非決定性的実行の起り得るもう一つの形は，ある特定の事象の系列が生ずると図式が永久に“停止”するかもしれない場合である（たとえば**図 II-8**）．この考えはペトリ・ネットの“生きていること”という概念と似ている．**図II-8**と**図II-1**との間の相似性を考えよ．これらは本質的には2つの異った図式すなわちペトリ・ネットと計算図式とを用いて同一のシステムをモデル化している．

II. 3. 3 プログラム図式に関する結論

ペトリ・ネットと計算図式とは，並列システムに対するたくさんのクラスのモデルのうちの2つの例に過ぎない．残念ながら，これらのモデルはオペレーティング・システムの設計者に対しては，直観的に訴える以外には直接的な利益はあまりもたらさなかった．しかし，計算する実際の関数を抽象化して資源の共有に関する特別な問題を集中して扱う，非計算的モデルがある．次節ではそのようなモデルを紹介する．

Ⅱ.4 デッドロックの問題に対するモデル

ペトリ・ネットおよび計算図式においては，デットロック状態に達するという問題があらわれてくる．デットロックを研究するための非計算的なモデルが，いくつか考案されてきた（たとえば Coffman ほか [1971]，Habermann [1969]，Hebalkar [1970]，Holt [1972]）．そのようなモデルの例として R. C. Holt の再使用型資源モデルを概説する [Holt, 1972]．

再使用型資源テデルにおいては，システムは**プロセス**の集合と一定の個数の**資源**（resource）の集合とから成ると考える．プロセスはいくつかの資源を**要求**（request）したり，**獲得**（acquire）したり，**解放**（release）したりして，システムの状態を変更する．図形的な形では，プロセスと資源とはそれぞれ四角と円とで表わす．四角の中には，その資源の存在する数に対応する石を置いておく．プロセスから資源へ向かう辺は，その資源に対する一単位の要求を示す．資源からプロセスへ向かう辺は，そのプロセスに対する資源の割り当てを示す．各資源の各単位は，一時には高々一つのプロセスにしか割り当てることができないので，資源から他のプロセスへの辺の総数は，その資源の総数を越えることはできない．

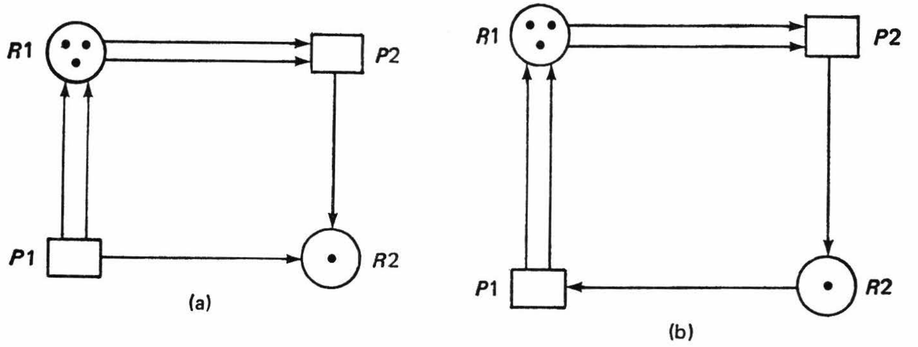


図 II-9 再使用型資源グラフの例. (a) 初期状態；(b) P_1がR_2を一単位獲得した後の状態.

図 II-9a は再使用型資源システムにおける一つの状態を示している．プロセス$P1$は$R1$を２単位，$R2$を１単位，要求している．$P2$ は$R1$ を２単位所有し，$R2$を１単位要求している．$P1$が要求している$R2$を１単位獲得したと仮定しよう．プロセスは要求した資源をすべて獲得しなければ，どれも解放しないという規則を採用すれば，$P1$ の要求を認めるとデッドロック状態を引き起す．$P1$ は$P2$が$R1$を１単位解放するまで実行を続けることはできず，また$P2$ は $P1$ が $R2$を解放するまで続けることはできない．

この形式化を用いれば，次に述べる規則に従ってグラフを**簡約**（reduce）することにより，デッドロックの状態を発見することができる．すべての停止していないプロセス，すなわち要求をすべて満すことが可能なプロセスに対して，そのプロセスから出たりそこへ入ったりしている辺をすべて削除する．グラフは簡約の後で，辺を一つも持たないとき**完全に簡約可能**（completely reducible）であるという．このモデルを用いたシステムは，それが完全に簡約可能なときデッドロックは生じないという定理がある．**図 II-10** では**図II-9a** を簡約している．**図 II-9b** においては，両方のプロセスが停止しているので，簡約は不可能である．

グラフにおいて，節点 a の**子孫**（progeny）とは a からの路の上に存在するすべての節点の集合をさす．**結び目**（knot）とは節点の集合Kにおいて，Kの

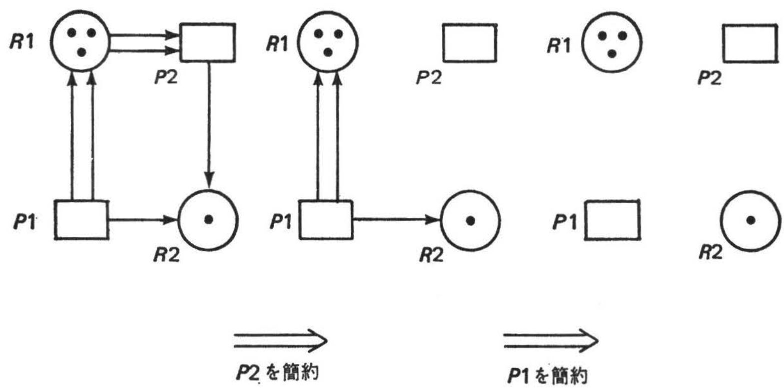


図 II-10　再使用型資源グラフの簡約化

各節点の子孫がちょうどKになるような節点の集合をさす．**図 II-11** において，{R1, R2, P1, P2} が結び目を構成している．要求を出しているすべてのプロセスが停止しているとき，状態は**臨界**（expedient）であるという．したがって，**図 II-11** は臨界である．Holt は臨界のシステムにおいては，結び目のあることがデッドロックに対する十分条件であることを証明した．

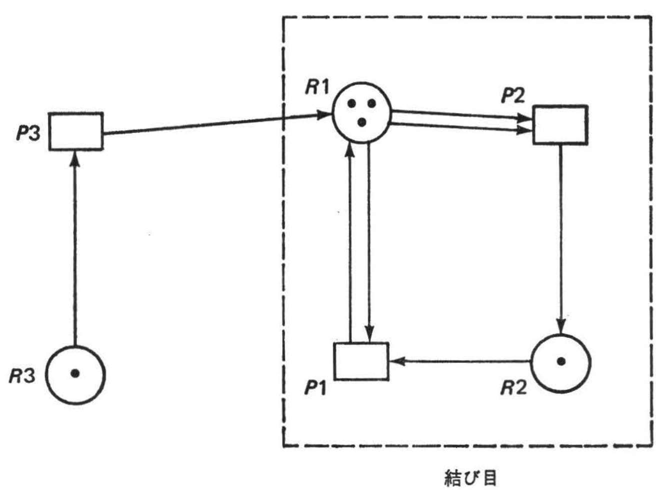


図 II-11　再使用型資源グラフのデッドロック．(P_1とP_2 は，結び目に入っているので，デッドロックの状態にある)．

メッセージの様にシステムの動きにつれて数が変化する資源は，**消費型**(consumable) という．消費型および再使用型資源の両方を用いれば，このモデルによって計算機システムの妥当な表現を作り出すことが可能である．このモデルを用いてデッドロックの実行時の検出と防止に対していくつかのアルゴリズムを提案している [Holt, 1972].

再使用型資源グラフはプロセスに対する資源の要求と割り当てを用いて，システムの状態をモデル化している．しかしながら，システムの状態は他の方法でもモデル化できる．たとえば，Holt は自分のモデルとペトリ・ネットとの関係づけを行ない，ペトリ・ネットの状態が資源グラフの状態をどのようにモデル化できるかということを示している．

今まで述べてきた Holt のモデルに関する議論は，少し乱暴なやり方であった．このモデルは，実際は非常に包括的であり，協力して働くプロセスの間でおこる興味のある動作をいろいろと記述することができる．ここでの議論は，スペースの関係で必然的に簡単になってしまった．モデルの詳細は Holt [1972] の文献によって与えられている．

II.5 おわりに

本書を通して，不当な形式化を避けるように意識して努めてきた．数学的モデルや結果は，重要な問題の解決に対する何らかの洞察を与えてくれるかヒントを与えてくれるときに，はじめて重要になるわけである．残念ながら，オペレーティング・システムにおける問題を主とし，またそれが動機で行なわれた理論的な結果の大部分は，すぐには適用可能ではない．これはそれらの結果が無視されるべきであるということを意味しているのではない．良い理論は，それ自身喜ばしいことである．それにより学生達は概念を視覚化し理解し易くなる．時には，新しいアイデアを生み出すきっかけとさえなる．

この補遺では，並列処理においてあらわれる問題に対して，何らかの洞察を与えてくれるようないくつかの理論的モデルを提示した．そのようなものとして，オペレーティング・システムにおいてプロセスの構造や相互関係を設計者がよく理解するのを，それらは助けることができる．

補　遺　III

おもちゃのオペレーティング・システム

III.1 は じ め に

オペレーティング・システムのコースでは，学生は原理と基本的な技法とを学ぶ．これらの技法が，実際にはどのように適用できるのかを理解するのには例題と演習が役に立つ．残念ながら，講義では充分ではない．いくつかの問題を考えている学生にとっては，自分自身でそれに立ち向かわなければならない．その学生が問題の重要性を理解したときには，彼は適当な題材を熱心に吸収するであろう．誰でも必要が生じたときに本当にわかるのである．学生に“おもちゃのオペレーティング・システム”を作らせる目的は，ソフトウェアの設計の実際の姿にいくらかでも立ち向かわせるという点にある．

システムを構成するときは，学生達は2～3のチームに編成する．ソフトウェアの設計プロジェクトにおいて，協力し合うということはそれ自体で一つの経験になる．学部の計算機科学のカリキュラムでは，学生達はたくさんプログラムを書くが，それらは通常何の原則もなく書けるような“おもちゃ”のプログラムである．おもちゃのオペレーティング・システムは，決しておもちゃのプログラムではない．それは組織化した方法で設計し，実現し，テストしなければならない複雑なプログラムである．学生は紙きれの前に座って，すべてのアイデアを一瞬のうちに得られることを期待してはならない．また，学生は本書のあちこちの章で述べてきた多くの題材を実際に理解せずに，オペレーティング・システム——たとえおもちゃであっても——を設計することはできない．

オペレーティング・システムにおける実際的な経験を学生に与えるときの主な問題点は費用のことである．彼等は大学内の商品としてのオペレーティング

・システムをむやみにいじることは許されていない．それはお金の点からみて非常に費用がかかり，他のユーザに提供するサービスを低下させてしまう．それゆえ，オペレーティング・システムの多くの問題をシミュレートするような環境を作り出さなければならない．そのような環境を実現する時に主として必要なことは，多重タスキングを持った PL/I のようにコルーチンや並列処理を許している言語を持つことである．Toronto 大学では，おもちゃのシステムは TOPPS 言語で書いているが，それは並列処理と基本的な同期基本命令とを提供してくれる ［Holt および Kinread, 1972］．TOPPS 言語はシステムの多重タスキングの能力（すなわち IBM OS/360）は用いず，むしろ逐語解釈的に多重プロセシングをシミュレートしている．これによりプログラムの費用を減少し，実際のシステムにおけるデッドロックの可能性を除去している．

課題はスプールのある多重プログラミングのシステムを，並列処理言語で書くことである．全体のシステムは協力して働くハードウェアの装置とソフトウェアのプロセスとの集団と考えることができる．ハードウェアの装置はプロジェクトの仕様の一部として学生に手渡したプログラムによってシミュレートする．オペレーティング・システムはハードウェアをシミュレートしている言語と同じもので書かれるので，“ハードウェア”とソフトウェアとのインターフェイスの問題は無視できる．ハードウェアとのインターフェイスの問題を除去することは課題の規模を小さくするための一つの鍵である．実際の機械では，ハードウェアの複雑さにより一学期コースで管理できるプロジェクトにはきっとならないであろう．

Toronto 大学で用いている課題は，R. C. Holt によって設計されたおもちゃのオペレーティング・システムのプロジェクトに基づいている．同様のプロジェクトが文献に発表されている ［Shaw および Weiderman, 1971］．

次節では，シミュレートする機械に対する提案を示す．これらの提案の多くのものは自分達の経験に基づいている ［Czarnick ほか, 1973］． 第III．3節ではシミュレートした機械に対し小さなオペレーティング・システムを書くときの基本的な問題を議論する．

Ⅲ.2 シミュレートしたハードウェア

シミュレートしたハードウェアの機械の主な特徴は，おもちゃのオペレーティング・システムに対し非常に明快なインターフェイスを提供することである．複雑な割り込みやプログラムのトラップの構造とかおかしな機械語を避けるためには，これは最善の方法である．同様に，各ハードウェア装置を開始するコマンドは単純にすべきである．単純でよく設計した機械に対してもシステムを書くことは充分難しい．実際の世界のハードウェアの複雑さは，興味のある実現上の問題をうみ出すけれども，そのためにまた一学期のコースに対しては大きすぎるプロジェクトになってしまう．

ハードウェア・システムは読取り機，印刷機， CPU， 主記憶，ドラムのような二次記憶装置から構成される．システムの中心は CPU であるが，それは主記憶に格納してあるユーザ・プログラムを実行する．簡単に解釈できる小型で単純な機械語はユーザ言語として十分であるけれども，もっと優雅なアセンブリ言語または高級言語を試みてもよい．主記憶は，並列処理言語では配列として表わすことができるが，そこにはユーザ・プログラムとデータとを格納する． CPU は命令カウンタ，汎用レジスタ，ページ・テーブル・レジスタ，ベス・リミット・レジスタなどそのすべての内部レジスタの値から成り立つ“プロセス状態語”を受け取ることにより起動される．そして CPU はプログラムが“割り込む”まで命令の実行を続ける．割り込みは，入出力要求とか，故意の不当動作によって生じたのであろう．この時点でオペレーティング・システムを呼び出し，適当な処置を行なわなければならない．呼び出しの方法は，並列処理言語に於て使用できる制御構造によって決まる．

読取り機や印刷機は，主記憶の番地と長さの項目を受け取ることにより起動する．ドラムは2つの番地と，どの方向にデータが移動するのかを指定する操作コードが必要である．これらの装置が入出力の終了をおもちゃのオペレーティング・システムに知らせる方法は，やはり使用する言語によってきまる．

この機械に対するオペレーティング・システムはユーザの言語ではなく，並列処理言語によって書いてある．それを具体的に考える一つの方法は，その並列処理言語が一つのユーザ・プロセス(すなわちシミュレートしている CPU)，

一つのシミュレートしている印刷機プロセス，一つのシミュレートしている読取り機プロセス，一つのシミュレートしているドラム・プロセス，それに任意個のシステム・プロセスを実現することである．ユーザ・プログラムは，一つのCPUプロセスの上でしか実行できない．すべてのシステム・プログラムは自分自身の仮想処理装置を持つことを許している．システム・プロセスはすべてのユーザの中で一つのユーザ・プロセスを多重プログラムしなければならない．

前述の機械は，提案の概要を紹介しただけである．論理的には，任意の妥当な機械のアーキテクチャが可能である．しかしコースの前後の関係からみて，複雑な機械は，ハードウェアでシミュレートしたプログラムをプロジェクトの仕様の一部として学生達に提供するときにのみ利用できる．それでも複雑な機械は複雑なシステムを必要としがちである．教育上は，プロジェクトを簡潔にしておくのは良いことであろう [Czarnick ほか，1973]．

Ⅲ.3 おもちゃのオペレーティング・システム

おもちゃのオペレーティング・システムに対する要求は，入力と出力をスプールし，ユーザのジョブを多重プログラミングの環境のもとで実行することである．実在するオペレーティング・システムの大部分は，少なくともこのような基本的能力すなわちスプーリングとか多重プログラミングを備えている．オペレーティング・システムは読取り機から入ってくるジョブを読み，それらをドラム上に置く．記憶空間が使用できれば，ジョブを多重プログラミングしている主記憶内に持ってくる．ジョブが入出力操作のために中断しているときは，他のジョブにCPUを与える．ジョブが印刷する出力は主記憶の中でバッファし，しばらくしてドラムの中に置く．最終的には，出力はドラムから印刷機へスプールする．当然これらの多くの処理を並列的に行なう．すなわち，ジョブはドラムやCPUの動作と並行に読み込み，書き出す．

オペレーティング・システムは多くの機能を実行しなければならない．新しいジョブをドラム上に読み込むこと，ドラムから完了したジョブを印刷して出力すること，ドラムから主記憶へジョブをロードすること，ユーザ・プログラムのトラップを処理すること， CPU に対してジョブをスケジュールすること，ジョブが完了したときに主記憶からジョブを追い出すことなどである．これら

の独立した機能に対する構造には多数の可能性がある．各機能は別々のプロセスでもよいし，いくつかの機能が集まって一つのプロセスになってもよい．たとえば，一つの組織の型では一つのプロセスを３つの周辺装置の各々に関連づけて，それらの装置に対する要求の待ち行列をつくる方式である．４番目のプロセスはユーザ・ジョブを見張り，CPUをスケジュールする責任をもつことができる．

Ⅲ.4 お わ り に

前節で述べたおもちゃのオペレーティング・システムのプロジェクトは，学生達に実際の設計の経験を得る機会を与えてくれる．学生達はシステムで使用可能な別々の資源やその割り当て方式について習熟するようになる．しかしながら，そのようなプロジェクトを実行する際には，背景にいくつかの問題がある［Czarnick ほか，1973］．

第１の問題は，費用のことである．各々の学生のチームではプロジェクトを完了させるために IBM 370/165 を約15分消費する可能性がある．読者はこの費用を自分の機械に外挿することができるだろう．大抵の標準からみれば，これはかなり高い．費用はシステムの複雑さを減らすことにより，幾分減少させることができる．各チームが消費してよい計算時間の量を制限するのもよいことかもしれない．どんな場合でも，オペレーティング・システムに対する実験をすることは非常に安くはならない．

プロジェクトに関する第２の問題は，複雑さである．多くの学部の学生達は一学期間のコースで，このプロジェクトを完成させるのは困難だと思っている．学生達には自分のシステムを簡潔にするよう指導してやるべきである．別の方法として，学生達にはプロジェクトの仕事の一部として非常に簡単な単一プログラムのシステムを与え，それを拡張してスプールのある多重プログラミング・システムのプロジェクトにするように要求する．この方法は，プロジェクトを走らせる費用を減少させるという余分の利益ももたらす．

第３の問題は，初期の努力である．プロジェクトを開始するには，いくらかの作業が必要になる．多くの計算機施設は並列処理言語を持っていないが，それは明らかに本質的に必要な要素である．また，この位の規模の課題は詳細な

仕様を必要とする．この課題を完全に記述すると10〜20ページにわたるのが普通であろう［Czarnick ほか，1973］．

第4の問題は，運営に関してである．この位のプロジェクトを監督するのにはかなりたくさんの時間がかかる．設計の質，ドキュメンテーション，必要な実行の回数などを含む全体の性能に対して，学生達を採点しなければならない．プロジェクトの採点は，また時間がかかる．このコースにおける助手はプログラムを良く読めなければならない．

Toronto 大学においては，学生達はこのプロジェクトに熱心であり，それが価値のある経験であると考えている．しかしながら，野心的なシステム設計をすると，時には学生に非常な負担をかけることになり，他のコースの勉強に影響したりコースの成績が遅れて事務をこまらせたりする．それにもかかわらず，プロジェクトの仕様がうまくでき，ハードウェアのシミュレーション・プログラムのような詳細な事実が事前に与えられれば，第III・3節に述べたような基本的システムは，おおよそ一学期のコースに相当する努力をすれば大部分の学生が完成することができる．

注釈つき参考文献一覧表

略語：

CACM Communication of the Association for Computing Machinery

JACM Journal of the Association for Computing Machinery

SJCC Spring Joint Computer Conference

FJCC Fall Joint Computer Conference

AFIPS American Federation of Information Processing Societies

Abate, J., and Dubner, H. [1969]. "Optimizing The Performance of a Drum-Like Storage," *IEEE Trans. Computers* 18, No. 11 (Nov.), 992-997.

各トラック毎にヘッドを一つ持つ固定ヘッド記憶装置について，最小回転所要時間先処理（shortest-rotational-latency-first）の待ち行列方式の下での解析をしている．この研究の動機はこのアルゴリズムによってスケジュールされる新しく導入されたハードウェア装置であった．システムを記述し，待ち行列の長さと待ち時間の平均値の解析的な推定が示されている．

Aho, A. V., Denning, P. J., and Ullman, J. D. [1971]. "Principles of Optimal Page Replacement," *JACM* 18, No. 1 (Jan.) 80-93.

ある仮定のもとでは，要求ページ化は他の任意のページ置き換え則と少なくとも同じ能率であることが示されている．次に，あるページ置き換え則の費用を調べるための理論的モデルが開発されている．

Alexander, C. [1964]. *Notes on the Synthesis of Form.* Harvard Univ. Press, Cambridge, Massachusetts.

この本は，大規模システム（たとえば町）を設計するときの建築家の見解を示している．システムをサブシステムに自然に分割することを強調している．

Alsberg, P. A., and Mills, C. R. [1969]. "The Structure Of the ILLIAC IV Operating System," *Proc. Second ACM Symp. Operating Systems Principles* (Oct.), 92-96.

ILLIAC IV 配列計算機に対するオペレーティング・システムを紹介している．Burroughs 社の B5500 がこのシステムに結合され，オペレーティング・システムの機能の大部分を実行している．新しいハードウェアをもとにしているので，このシステムの記述は特に面白いものとなっている．

Atwood, J. W. (ed.), **Clark, B. L., Grushcow, M. S., Holt, R. C., Horning, J. J., Sevcik, K. C., and Tsichritzis, D.** [1972]. *Project SUE Status Rep.*, Tech. Rep. CSRG-11. Comput. Systems Res. Group, Univ. of Toronto, Toronto.

これは，プロジェクト SUE を紹介した最初の大きなドキュメントである．システムはこのレポート以来，大幅に発展し，この内容のあるものは古い話となってしまっている．

Baer, J. L. [1973]. "A Survey of Some Theoretical Aspects of Multiprocessing," *ACM Comput. Surveys* **5**, No. 1 (Mar.), 31-80.

多重処理システムに対するプログラミング言語の特性，理論的モデル，性能の解析について概括している．大量の参考文献とハードウェアの多重処理に対する付録がついている．

Baker, F. T. [1972]. "Chief Programmer Team Management of Production Programming," *IBM Systems J.* **11**, No. 3, 56-71.

プロジェクト管理に対する主任プログラマ制度の技法を *New York Times* 情報バンク・プロジェクトと関連づけて議論している．

Ballard, A. J., and Tsichritzis, D. [1973]. "Structure and Correctness of Systems," *Proc. Canad. Comput. Conf.*, Edmonton (June), 324-340.

大規模システムの検証における特別な問題を議論している．オペレーティン

グ・システムの正当性に対する進め方についても概略を示している.

Bard, Y. [1971]. "Performance Criteria And Measurement In A Time-Sharing System," *IBM Systems J.* **19**, No. 3, 193-216.

CP-67オペレーティング・システムを修正の前と後とで評価するために統計的なモデルを議論している. この方法により変更の効果を測定し, 効果の集団からその効果を区別することができる.

Belady, L. A. [1966]. "A Study of Replacement Algorithms For A Virtual Storage Computer," *IBM Systems J.* **5**, No. 2, 78-101.

ページ置き換えアルゴリズムを, プログラムについて必要とする情報の量に基づいて分類している. 最適アルゴリズム (Mattson ほか, [1970] も参照のこと) を述べ, いくつかのアルゴリズムに対するシミュレーションの結果を議論している.

Belady, L. A., and Kuehner, C. J. [1969]. "Dynamic Space-Sharing In Computer Systems," *CACM* **12**, No. 5 (May), 282-288.

記憶空間と処理装置の時間との兼ね合いを解析する方法を提案している. 2つのページ置き換えアルゴリズムを, この方式を用いて比較している. (第5.4.4.1節参照)

Belady, L. A., and Lehman, M. M. [1971]. *Programming System Dynamics or the Meta-Dynamics of Systems in Maintenance and Growth*, No. RC3546, IBM Res. Center, Yorktown Heights, New York (Sept.)

大規模なプログラミング・システムの成長のモデルを提案している. 時間と共にソフトウェアは修正され, したがってたんたん複雑になりそれを保守するのが困難となる. 著者等はいくつかの大規模ソフトウェア・システムにおいては, 保守の努力が指数関数的に増加することが避けられないと主張している. これらのシステムにおいては, 有限の生存期間しか保証していないことをこれは意味している.

Belady, L. A., Nelson, R. A., and Shedler, G. S. [1969]. "An Anomaly In Space-Time Characteristics of Certain Programs Running In A Paging Environment," *CACM* **12,** No. 6 (June), 349–353.

FIFO ページ置き換え則に対し記憶の割り当てを増やすと，異常な参照パターンのもとではページ・フォールトの数が増える可能性のあることを示している．Mattson 等 [1970] は "スタック" アルゴリズムでは，このような FIFO 効果が生じないことを示した（第5.4.2.2節も参照せよ）．

Bell, C. G., and Gold, M. M. [1971]. "An Introduction to the Structure of Time-Shared Computers," *Advances in Information Sci.* **4,** 161–272.

時分割計算機システムのハードウェアと時分割オペレーティング・システムの構造と外部特性とを議論している．例として，PDP-11 の事例研究を行なっている．

Bernstein, A. J., and Sharp, J. C. [1971]. "A Policy-Driven Scheduler For A Time-Sharing System," *CACM* **14,** No. 2 (Feb.), 74–78.

紹介しているスケジューリング・アルゴリズムは，各ユーザに与えられた方針関数に基づいている．各関数は，どれだけ多くの処理をユーザが受けるべきかを，時間の関数として指定している．そして，ユーザの優先度は方針関数による処理の期待値と実際に得られた処理との差に従って変動する．このアルゴリズムを解析しその特別な実現を議論している．

Bernstein, A. J., Detlefsen, G. D., and Kerr, R. H. [1969]. "Process Control And Communication," *Proc. Second ACM Symp. Operating Systems Principles* (Oct.), pp. 60–66.

事象の待ち行列を用いたプロセス間の通信の機構を述べている．事象の操作は CAUSE と MODIFY およびその逆操作である UNCAUSE と DELETE である．この機能は第2章で議論したものとは別のものとして興味がある．

Bernstein, P. A. [1973]. *Description Problems in the Modeling of Asynchronous Computer Systems,* Tech. Rep. No. 48. Dept. of Comput.

Sci., Univ. of Toronto, Toronto.

12個の並列システムに対するモデルを紹介し，その記述能力にもとづいて比較している．このレポートはプログラム図式に関する膨大な参考文献を含んでいる．

Brawn, B. S., and Gustavson, F. G. [**1968**]. "Program Behavior In A Paging Environment," *Proc. AFIPS* **33**, Pt. II, *FJCC*, pp. 1019–1032.

自動的な記憶管理をユーザの立場から調べている．この研究の結果によれば，ページ化の全体の効率に関して，置き換えアルゴリズムよりもプログラミングの様式の方がはるかに重要であると提言している．

Brinch Hansen, P. [**1970**]. "The Nucleus Of A Multiprogramming System," *CACM* **13**, No. 4 (Apr.), 238–241.

この論文は RC 4000 の核を紹介している．システムは郵便受けを介して連絡し合う，協力して働くプロセスの階層となっている．このシステムはいろいろのオペレーティング・システムを構築するためのソフトウェアの基礎を提供している（第1. 4節も参照）．

Brinch Hansen, P. [**1971**]. "Short-Term Scheduling In Multiprogramming Systems," *Proc. Third ACM Symp. Operating Systems Principles* (Oct.), pp. 101–105.

多重レベル・スケジューリング方式における，短期スケジューラに対するアルゴリズムをいくつか提示している（第3. 4節参照）．このアルゴリズムは詳細に述べられており，実際の実現に自然に到達できる．

Buxton, J. N., and Randell, B. (**eds.**) [**1970**]. *Software Engineering Techniques.* NATO Sci. Affairs Div., Brussels.

このレポートはソフトウェア工学に関する1969年 NATO 会議の記録である．このレポートはプログラムの正当性，ソフトウェアの信頼性，いくつかの大規模システムの事例研究の内容を含んでいる．

Calingaert, P. [**1967**]. "System Performance Evaluation: Survey and Appraisal," *CACM* **10**, No. 1 (Jan.), 12–18.

計算機システムの性能の目標を定義し，解析，シミュレーション，合成などの基本的技法を議論している．

Clark, B. L., and Horning, J. J. [1971]. "The System Language for Project SUE," *ACM SIGPLAN Notices* **6**, No. 9 (Oct.), 79-88.

PASCAL に非常に似ているシステム記述用言語としての SUE システム言語をこの論文で紹介している．

Coffman, Jr., E. G. [1969]. "Analysis of a Drum Input/Output Queue under Scheduled Operation in a Paged Computer System," *JACM* **16**, No. 1 (Jan.), 73-90.

補助記憶として働いているドラムに対して，使用率，平均待ち行列長さ，平均待ち時間を測定するために数学的モデルを用いている．

Coffman, E. G., Jr. and Denning, P. J. [1973]. *Operating System Theory*. Prentice-Hall, Englewood Cliffs, New Jersey.

オペレーティング・システムにおける問題のモデル化や解析について，いろいろの角度から見て提示している．資源割り当て問題の解析的な扱い方に関する良い参考書にもなる．

Coffman, Jr., E. G., and Kleinrock, L. [1968]. "Computer Scheduling Measures And Their Countermeasures," *Proc. AFIPS* **32**, *SJCC*, pp. 11-21.

ジョブ・スケジューリングの主要な方法を概括し，名方法の弱点を利用するユーザの戦略も述べている．

Coffman, Jr., E. G., and Randell, B. [1971]. "Performance Prediction for Extended Paged Memories," *Acta Informat.* **1**, Fasc. 1, 1-13.

著者等はページ化された記憶の大きさを拡張するためのモデルを開発している．プログラムのページ参照の軌跡を与えると，彼等の解析的技法を用いれば，コア内のページ・フレームの数が増加したときに，ページ化アルゴリズムの性能を予測することができる．

Coffman, Jr., E. G., and Ryan, Jr., T. A. [1972]. "A Study of Storage Partitioning Using a Mathematical Model of Locality," *CACM* **15**, No. 3 (Mar.), 185-190.

ページ化の環境のもとで，分割の大きさに対する数学的モデルを提示している．ワーキング・セットの大きさの変動が“小さい”とき，固定分割法は容認できることを示している．

Coffman, Jr., E. G., and Varian, L. C. [1968]. "Further Experimental Data On The Behavior Of Programs In A Paging Environment'" *CACM* **11**, No. 7 (July), 471-474.

異った置き換えアルゴリズムのもとで，いくつかのプログラムのページ・フォールトの振舞について実験的結果を述べている（第5.4.4.1節参照）．

Coffman, Jr., E. G., Elphick, M., and Shoshani, A. [1971]. "System Deadlocks," *ACM Comput. Surveys* **3**, No. 2 (June), 67-78.

この概括は，順次的に再使用可能な資源を持つシステムにおけるデッドロックについて行なわれた主要な研究のいくつかを紹介している．プロセスの資源の要求が事前に正確にわかっている場合，プロセスの資源の要求の最大値だけがわかっている場合，資源の要求がそれを要求されるまでわからないような場合，これらの場合に対するデッドロックの検出を扱っている．

Comeau, L. W. [1967]. "A Study of the Effect of User Program Optimization in a Paging System," *Proc. ACM Symp. Operating System Principles*, 7 pp.

ページ化の振舞に対するユーザ・プログラムの影響の実験からみると，ユーザによる最適化がページ化システムの全体の性能に対して大きな改善をもたらすことがわかった．

Conway, M. E. [1963]. "Design of a Separable Transition Diagram Compiler," *CACM* **8**, No. 7 (July), 396-408.

COBOL コンパイラを設計する方法を議論している．この論文はコルーチンという概念を導入したということで，特に有名である．

Conway, M. E. [1968]. "How do Committees Invent ?," *Datamation* **14,** No. 4 (Apr.), 29–32.

システムの構造はそれを作り出す組織の構造に似ているという説を提示している．著者は，またこの考え方が設計グループの構造に対してもつべき影響についても議論している．

Conway, R. W., Maxwell, W. L., and Morgan, H. L. [1972]. "On the Implementation of Security Measures in Information Systems," *CACM* **15,** No. 4 (Apr.), 211–220.

保全性行列モデルを定義し，実在するいくつかの保全性システムをモデル化するのには十分強力であることを示している．

Corbato, F. J. [1969]. "PL/I As A Tool For System Programming," *Datamation* **15,** No. 5 (May), 68–76.

MULTICS プロジェクトに対するプログラミング言語として，PL/I を選んだ理由を議論している．この論文は第9．2節の補足的な読みものとして良い．

Corbato, F. J., and Saltzer, J. H. [1969]. "Some Considerations Of Supervisor Program Design For Multiplexed Computer Systems," *Proc. IFIP Congr.* 1968, pp. 315–321. North-Holland Publ., Amsterdam.

多重化に関する問題および情報の共有によって生じてくる問題について議論している．時分割システムを特に扱っている．

Courtois, P. J., Heymans, F., and Parnas, D. L. [1971]. "Concurrent Control with 'Readers' and 'Writers'," *CACM* **14,** No. 10 (Oct.), 667–668.

（第２章の演習問題で述べた）読み書き問題に関する２つの解答を示している．

Czarnik, B. [editor], **Tsichritzis, D., Ballard, A. J., Dryer, M., Holt, R. C., Weissman, L.** [1973]. *A Student Project for an Operating System Course*, CSRG-29, Computer Systems Research Group, Univ. of Toronto.

この技術レポートは，000/Z おもちゃのオペレーティング・システム・プロ

ジェクトの記述と，TOPPS 並列処理言語のユーザ・マニュアルとから成る．TOPPS で書かれた 000/Z の例も示してある．

Dahl, O.-J., Dijkstra, E. W., and Hoare, C. A. R. [1972]. *Structured Programming*, Academic Press, New York.

この本は構造的プログラミングに対する3つのエッセイから成る．第一の論文は Dijkstra の"構造的プログラミングに関するノート"であるが，これは下降的プログラム設計の組織的な方法を提示している．Hoare の"データ構造に関するノート"は，データ構造に適用した同様の概念を提示している．第三の論文では，Simula 67 の言語の中で前述したような概念を合成している．

Denning, P. J. [1968]. "The Working Set Model For Program Behavior," *CACM* **11**, No. 5 (May), 323–333.

これはワーキング・セット・モデル（第5.4.1節参照）の主要な概念を概説した Denning の最初の論文である．

Denning, P. J. [1970]. "Virtual Memory," *ACM Comput. Surveys* **2**, No. 3 (Sept.), 153–190.

この概括では，仮想記憶システムのほとんどの特徴について深い議論を展開している．この分野に関する完全な参考文献も含んでいる．

Denning, P. J., and Schwartz, S. C. [1972]. "Properties of the Working Set Model," *CACM* **15**, No. 3 (Mar.), 191–198.

ワーキング・セットに関するいくつかの数学的考察，たとえばワーキング・セットの大きさとミッシング・ページ率との関係，ワーキング・セットの大きさの分布，ワーキング・セットの大きさへのページ間の依存度の影響などを議論している．

Denning, P. J., Dennis, J. B., Lampson, B. W., Habermann, A. N., Muntz, R. R., and Tsichritzis, D. [1971]. *An Undergraduate Course on Operating System Principles.* Cosine Committee on Education of the Nat. Acad. of Engrng., Washington, D. C.

このレポートは COSINE 作業グループⅧが書いたが，オペレーティング・

システムの完全なコースを紹介している．本書の基本構成はこの概括に従って組み立てられた．

Dennis, J. B. [1965]. "Segmentation And The Design of Multiprogrammed Computer Systems," *JACM* **12**, No. 4 (Oct.), 589–602.

いくつかの異った番地付け機構を名前空間，記憶空間，番地空間の立場で議論している．共有，保護，動的割り当てなどの立場から番地機構の柔軟性を強調している．

Dennis, J. B. [1969]. "Programming Generality, Parallelism And Computer Architecture, ," *Proc. IFIP Congr. 1968*, pp. 484–492. North-Holland Publ., Amsterdam.

システムの並列性をプログラム図式の観点から議論している．

Dennis, J. B. [1971]. "Coroutines and Parallel Computation," *Proc. Fifth Annu. Princeton Conf. Information Sci. and Systems*, pp. 293–294.

コルーチンの概念 [Conway, 1963] を並列に実行するプロセスの立場で議論している．

Dennis, J. B., and Van Horn, E. C. [1966]. "Programming Semantics For Multiprogrammed Computations," *CACM* **9**, No. 3 (Mar.), 143–155.

これは，多重プログラミング・システムにおける術語や概念を紹介した最初の論文である．資格リストと階層的ディレクトリを強調している．

Dijkstra, E. W. [1965]. "Solution to a Problem in Concurrent Programming Control," *CACM* **8**, No. 9 (Sept.), 569.

Dekker の2個のプロセスに対する解の拡張である n 個のプロセスの相互排除の解をこの論文で提示している．

Dijkstra, E. W. [1968a]. "Cooperating Sequential Processes," in *Programming Languages* (F. Genuys, ed.), pp. 43–112. Academic Press, New York.

現在では，プロセスの同期に関して古典的となっているこの論文で腕木，危

険部分の問題，デッドロックの除去を議論している．

Dijkstra, E. W. [1968b]. "The Structure Of The T. H. E. Multiprogramming System," *CACM* **11**, No. 5 (May), 341-346.

この論文は，上昇的設計の方式と腕木の定義も含めて THE システムを簡単に紹介している．

Dijkstra, E. W. [1968c]. "Go To Statement Considered Harmful," *CACM* **11**, No. 3 (Mar.), 147-148.

この有名な"編集者への手紙"において，Dijkstra は GO TO文に対する彼の基本的な反対の立場を概説している．

Dijkstra, E. W. [1971]. "Hierarchical Ordering of Sequential Processes," *Acta Informat.* **1**. Fasc. 2, 115-138.

非決定性的なハードウェアの相互作用を同期のとれた仮想機械に変換するときの問題と関連づけて腕木と秘書(すなわち Hoare のモニタ)とを議論している．

Doherty, W. J. [1970]. "Scheduling TSS/360 For Responsiveness," *Proc. AFIPS* **37**, *FJCC*, pp. 97-111.

TSS/360 の表引きスケジューラにおいて，パラメータを注意深く組織的に調整した結果，性能が驚くほど改善した．調整はワーキング・セットの大きさと現在のコアの割り当てに対して行なった．この研究に対する動機，調整それ自身，性能の結果を詳しく議論している．

Donovan, J. J. [1972]. *Systems Programming*. McGraw-Hill, New York. 訳書：池田克夫訳・**システム・プログラム** [1974]，日本コンピュータ協会

この教科書はアセンブラ，リンケージ・エディタ，ローダ，コンパイラ，オペレーティング・システムにおけるプログラミングの諸問題を IBM 流のやり方で展開している．

Earl, D. P., and Bugely, F. L. [1969]. "Basic Time-Sharing: A System of Computing Principles," *Proc. Second ACM Symp. Operating Sys-*

tems Principles (Oct.), pp. 75–79.
この論文はオペレーティング・システムのいくつかの定義とか，設計原理とかを組織化する試みを述べている．

Easton, W. B. [**1971**]. "Process Synchronization Without Long-Term Interlock," *Proc. Third ACM Symp. Operating Systems Principles* (Oct.), pp. 95–100.
ファイル・システムの危険なデータの各ブロックには"バージョン数"を与えておく．ブロックにアクセスする前後に，バージョン数をチェックしなければならない．このやり方を用いれば，長期間にわたりファイルからプロセスをしめ出す必要がなくなることを示している．

Elspas, B., Green, M. W., and Levitt, K. N. [**1971**]. "Software Reliability," *Computer* **4,** No. 1 (Jan.–Feb.), 21–27.
言語の設計とプログラムの検証に関する基本的問題を，ソフトウェアの信頼性という立場から議論している．

Elspas, B., Levitt, K. N., Waldinger, R. J., and Wakeman, A. [**1972**]. "An Assessment of Techniques for Proving Program Correctness," *ACM Comput. Surveys* **4,** No. 2 (June), 97–147.
この広範囲にわたる概括においては，プログラムの正当性の証明に関して，手で行なう場合と自動的に行なう場合の両者に対し，理論的背景と実用的考察とを議論している．

Fabry, R. S. [**1973**]. "The Case for Capability-Based Computers," *Fourth Symposium on Operating System Principles* (Oct.).
この論文は計算機システムの番地として，資格を自由に使用することを提案している．そのような構造が，何故役に立つかを議論した後で，資格に基づいた機械に対する実現の考察を議論している．

Floyd, R. W. [**1971**]. "Toward the Interactive Design of Correct Programs," *Proc. IFIP 1971,* BookletTA-1, pp. 1–4.
プログラマと会話型プログラム検証システムとの間の会話を仮定している．

著者は可能性のあるプログラム検証システムにおいて，どんなことができるのかを概説して結論としている．

Fontao, R. O. [1971]. "A Concurrent Algorithm For Avoiding Deadlocks," *Proc. Third ACM Symp. Operating Systems Principles* (Oct.), pp. 72–79.

処理しているプロセスと並列に実行するデッドロックの防止アルゴリズムを紹介している．このアルゴリズムは資源の要求や解放が実行されるごとに更新されるような安全状態の表を維持している．

Foster, C. C. [1971]. "An Unclever Time-Sharing System," *ACM Comput. Surveys* **3**, No. 1 (Mar.), 23–48.

CDC 3600 に対する小型の時分割システム UMASS-2 を紹介している．

Fuchs, E., and Jackson, P. E. [1970]. "Estimates of Distributions of Random Variables For Certain Computer Communications Traffic Models," *CACM* **13**, No. 12 (Dec.), 752–757.

ユーザと計算機との相互作用に対するモデルを提案し，実際のシステムからの測定値と一致して振舞うことを示している．

Fuller, S. H., and Baskett, F. [1972]. *An Analysis of Drum Storage Units*, Tech. Rep. No. 26. Digital Systems Lab., Stanford Electron. Lab. Stanford Univ., Stanford, California (Aug.).

確率的モデルを用いて，FIFO と最小遅れ時間先処理のドラム・スケジューリング・アルゴリズムを解析している．

Gaines, R. S. [1971]. "An Operating System Based On The Concept Of A Supervisory Computer," *Proc. Third ACM Symp. Operating Systems Principles* (Oct.), pp. 17–23.

SUE の核に似た CDC 6600 上に実現したオペレーティング・システムに対する小型のスーパーバイザーを紹介し，それを評価している．

Gear, C. W. [1969]. *Computer Organization and Programming.* Mc

Graw-Hill, New York.

この教科書はハードウェア，入出力，アセンブラ，分類と探索，コンパイル技法に対する入門書である．

Gotlieb, C. C., and Borodin, A. [1973]. *Social Issues in Computing.* Academic Press, New York.

この本は計算機が社会に与えた影響に関する広い範囲の問題を扱っている．特に，プライバシーに関する章は，第7. 6節にとって重要である．

Gotlieb, C. C., and MacEwan, G. H. [1970]. "System Evaluation Tools," in *Software Engineering Techniques* (J. N. Buxton, and B. Randell, ed.), pp. 93–99. NATO Sci. Affairs Div., Brussels.

システム設計のいろいろの場面におけるモデル化と監視の適用を議論している．

Graham, G. S., and Denning, P. J. [1971]. *Protection: Principles and Practice,* Tech. Rep. No. 101. Dept. of Elec. Enging., Princeton Univ., Princeton, New Jersey (Nov.).

保護に対するアクセス行列モデルを議論している．このモデルのオペレーティング・システムの設計に対する意味も扱っている．

Graham, R. M. [1968]. "Protection In An Information Processing Utility," *CACM* **11**, No. 5 (May), 365–369.

"リング" と呼ぶレベルを用いた，MULTICS の保護機構に対する抽象的モデルを紹介している．

Graham, R. M., Clancy, G. J., and DeVancey, D. B. [1971]. "A Software Design and Evaluation System," *Proc. ACM/SIGOPS Workshop on System Performance Evaluation,* pp. 200–213.

この論文はシステム設計，評価，実現のための言語 DES を紹介している．このシステムの主要な特徴は設計の過程において，シミュレーションにより，性能の推定ができる能力を持っていることである．

Gries, D. [**1971**]. *Compiler Construction for Digital Computers.* Wiley, New York. 訳書：牛島和夫訳・**コンパイラ作成の技法** [1976], 日本コンピュータ協会.

この本はコンパイラ作成の入門書であり，構文解析，コード生成および全体的な構成の考察を扱っている．

Habermann, A. N. [**1969**]. "Prevention of System Deadlocks," *CACM* **12**, No. 7 (July), 373–377, 385.

デッドロックの防止に対するいくつかのアルゴリズムを，資源割り当ての状態モデルと関連させて紹介している．特に，安全な状態と安全な遷移とを形式的に定義している．

Habermann, A. N. [**1972**]. "Synchronization Of Communicating Processes," *CACM* **15**, No. 3 (Mar.), 171–176.

プロセスの同期の機構を形式化するために，不変関係を用いている．この形式化を用いて連絡し合うプロセスのいくつかの性質を証明している．

Havender, J. W. [**1968**]. "Avoiding Deadlock In Multitasking Systems," *IBM Systems J.* **7**, No. 2, 74–84.

IBM OS/360 MVT において，デッドロックを避けるために用いた方法を提示している．

Hebalkar, P. G. [**1970**]. "Coordinated Sharing of Resources in Asynchronous Systems," *Record Project MAC Conf. Concurrent Systems and Parallel Comput., Woods Hole, Massachusetts*, pp. 151–168.

ペトリ・ネットに似たモデルを用いて，プロセスの振舞が事前に部分的にわかっているときに，システム内のデッドロックの性質を調べている．

Hoare, C. A. R. [**1969**]. "An Axiomatic Approach to Computer Programming," *CACM* **12**, No. 10 (Oct.), 576–580.

プログラムに対する公理の集合と推論規則とを定義している．これらの概念を単純なプログラムの手による検証に適用している．

Hoare, C. A. R. [1971]. "Proof of a Program: FIND," *CACM* **14,** No. 1 (Jan.), 39-45.

探索アルゴリズムの正当性の検証において，著者はプログラムの正当性の手による証明に対する一般的な方法を開発している．

Hoare, C. A. R., and Perrot, R. H. (**eds.**), [1972]. *Operating Systems Techniques.* Academic Press, New York.

この本は1971年に Belfast で行なわれたオペレーティング・システムの技法に関する国際セミナーの会議録である．議論の抜粋も含んでいる．

Hoffman, L. J. [1969]. "Computers and Privacy: A Survey," *ACM Comput. Surveys* **1,** No. 2 (June), 85-104.

プライバシーに関する法律的および技術的な問題を議論している．この文献には，大規模で注釈のついた参考文献一覧表がついている．

Holt, A. W., and Commoner, F. [1970]. "Events and Conditions," *Record Project MAC Conf. Concurrent Systems and Parallel Comput., Woods Hole, Massachusetts,* pp. 3-52.†

ペトリ・ネットを紹介し，解析し，並列操作を含むいくつかの簡単な問題に適用している．

Holt, R. C. [1972]. "Some Deadlock Properties of Computer Systems," *ACM Comput. Surveys* **4,** No. 3 (Sept.), 179-195.

いくつかのグラフ理論的モデルを用いて，デッドロックの検出と防止に対するアルゴリズムを開発している（第2. 7節および第Ⅱ. 3節参照）.

Holt, R. C., and Kinread, R. [1972]. "Teaching and Using High Level Concurrent Programming," *Proc. Canad. Comput. Conf., Montreal,* pp. 214202-214212.

並列処理を許す高級言語 TOPPS を紹介している．TOPPS を用いて，著者等は並列プログラミングの基本概念をどのようにして教えることができるのか

† 訳注． ACM より出版されている．

を議論している.

Horning, J. J., and Randell, B. [1973]. "Process Structuring," *ACM Comput. Surveys* 5, No. 1 (Mar.), 5–30.

この論文は並列プロセスに関連する多くの術語の精密な定義を提示している. 仮想機械とか抽象化のレベルのようなシステム構造における多くの概念も考慮している.

Huberman, B. J. [1970]. *Principles of Operation of the Venus Microprogram*, MITRE Tech. Rep. MTR-1843. The MITRE Corp., Bedford, Massachusetts.

Venus システムのマイクロプログラムされたハードウェアの仮想機械に関する詳細をこのレポートで述べている.

IBM System/360 Operating System: Concepts and Facilities [1965]. Form C28-6535-0. Data Processing Div., IBM Corp. White Plains, New York.

これは IBM/360 ファミリー機械のオペレーティング・システムの入門的なドキュメントである.

Ichbiah, J. D., and Rissen, J. P. [1971]. *Directions de Travail pour un Atelier de Software: Rapport Préliminaire*, Compagnie Internationale Pour l'Informatique, Paris (Oct.), pp. 3–9.

著者は信頼性のあるソフトウェアに対する道具を作るソフトウェア実験室の環境を提案している.

Karp, R. M., and Miller, R. E. [1969]. "Parallel Program Schemata," *J. Comput. System Sci.* 3, No. 2 (May), 147–195.

このかなり難解な論文は第 II. 2 節に紹介したものに似た並列計算のモデルを提示している. これはプログラム図式に関する重要な論文の一つである.

Keefe, D. D. [1968]. "Hierarchical Control Programs For Systems Evaluation," *IBM Systems J.* 7, No. 2, 123–133.

システム・プログラムをテストする方法を議論しているが，そこではシステムは非特権モードで実行し，特権モードで実行する特別なモニタがプログラムの入出力を処理しデータを集める．

Kernighan, B. W. [1969]. "Optimal Segmentation Points For Programs," *Proc. Second ACM Symp. Operating Systems Principles* (Oct.), pp. 47–53.

節点が，命令かデータ項目をあらわし，有向枝がその間の遷移をあらわすようなグラフとして，プログラムをモデル化する．ある仮定の下でページ化の振舞に対し，プログラム内の最適なブレーク・ポイントを見つけ出す線形時間アルゴリズムを紹介している．

Kernighan, B. W., and Plaugher, P. J. [1973]. "Programming Style for Programmers and Language Designers," *Record IEEE Symp. Comput. Software Reliability*, New York, pp. 148–154.†

良い様式を用いて書いたプログラムは悪い様式で書いたプログラムより良いプログラムになる可能性が大きい．プログラミング様式のいくつかの原理と例とをプログラムの信頼性に対する効果という観点から議論している．

Kilburn, T., Edwards, D. B. G., Lanigan, M. J., and Sumner, F. H. [1962]. "One-level Storage System," *IRE Trans. Electron. Comput.* **EC-11**, No. 2 (Apr.), 223–234.

この論文は Atlas 機械の簡単な紹介をし，コアとドラムを用いて仮想記憶を構成する一般的な方法を述べている．

King, J. C. [1969]. *A Program Verifier.* Ph. D. Thesis, Dept. of Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, Pennsylvania.

著者は検証をするコンパイラの構成について述べているが，それはコードをコンパイルする上に，プログラマの与えた命題がプログラムが実際に行なうことと一貫性がとれているかをどうかをチェックする．

---

† 訳注．これは次に示す単行本として出版されている．
Kernighan, B. W., and Plaugher, P. J. "The Elements of Pr gramming Style", McGraw-Hill, New York, 1974.

Kleinrock, L. [1970]. "A Continuum Of Time Sharing Scheduling Algorithms," *Proc. AFIPS* **36**, *SJCC*, pp. 453-458.
線形の優先順位の関数の2つのパラメータを変えることにより，スケジューリング・アルゴリズムのクラスが得られる．一つのアルゴリズム，たとえば自己中心的なラウンド・ロビンの完全な解析を示している．

Knuth, D. [1966]. "Additional Comments on a Problem in Concurrent Programming Control," *CACM* **9**, No. 5 (May), 321-322.
Knuth は n 個のプロセスの相互排除問題に対する Dijkstra の解 [Dijkstra, 1965] の問題点を述べそれに対する解を提案している．

Knuth, D. [1968]. "Fundamental Algorithms," *The Art Of Computer Programming*, Vol. I. Addison Wesley, Reading, Massachusetts.
このすばらしい本の後半は，データ構造とその処理を完全に扱っている（補遺Iを参照）．

Kuck, D. J., and Lawrie, D. H. [1970]. "The Use And Performance Of Memory Hierarchies: A Survey," in *Software Engineering*, Vol. 1, pp. 45-78. Academic Press, New York.
記憶の階層を構造，費用の節約，プログラムの振舞，全体のシステムの性能に対する影響などの観点から議論している．いくつかの実験的結果を引用している．

Lampson, B. W. [1968]. "A Scheduling Philosophy For Multiprocessing Systems," *CACM* **11**, No. 5 (May), 347-360.
資格，2レベルのスケジューリング，スケジューリング・アルゴリズムのハードウェアによる実現などを含んだ保護とプロセスのスケジューリングに関する基本的概念を総括している．

Lampson, B. W. [1969]. "Dynamic Protection Structures," *Proc. AFIPS* **35**, *FJCC*, pp. 27-38.
理論的および実用的側面を考えながら領分，資格，アクセス・キーを含むよ

うな一般的なモデルを概説している.

Lampson, B. W. [1971]. "Protection," *Proc. Fifth Annu. Princeton Conf. Information Sci. and Systems*, pp. 437-443.

この論文は保護に関する重要な抽象概念の大部分のものを展開している．領分，アクセス行列，資格，記憶保護に対する保護モデルを議論している．

Lewis, P. A. W., and Shedler, G. S. [1971]. "A Cyclic-queue Model Of System Overhead In Multiprogrammed Computer Systems," *JACM* **18,** No. 2 (Apr.), 199-220.

要求ページ化における多重プログラミング・システムの複雑な確率モデルを紹介している.

Liptay, J. S. [1968]. "Structural Aspects Of The System/360 Model 85: II The Cache," *IBM Systems J.* **7,** No. 1, 15-21.

IBM 360/85 に対するキャッシュ記憶，すなわちコアと CPU の間にある高速のバッファ記憶を紹介している.

Liskov, B. H. [1972]. "The Design Of The Venus Operating System," *CACM* **15,** No. 3 (Mar.), 144-149.

この論文は Venus システムのソフトウェアの本質的な特性を記述している（第10章参照).

London, R. L. [1970]. "Proving Programs are Correct: Some Techniques and Examples," *BIT* **10,** 168-182.

プログラムの正当性の証明を手で行なうための五つの技法を，概説し，各方法に対する例を提示している.

Lucas, H. C. [1971]. "Performance Evaluation and Monitoring," *ACM Comput. Surveys* **3,** No. 3 (Sept.), 79-91.

この論文は良く知られている評価の技法の大部分を，技術的でない見地から概括している.

Luconi, F. L. [1968]. *Asynchronous Computational Structures*, MAC-

TR-49. Ph. D. Thesis Dept. of Elec. Engrng., M. I. T., Cambridge, Massachusetts.

並列システムに対する“計算図式”を提示している．決定性とプロセスの相互通信とに関するいくつかの性質に対して，理論的扱いをしている．

MacDougall, M. H. [1970]. “Computer System Simulation: An Introduction,” *ACM Comput. Surveys* **2**, No. 3 (Sept.), 191–209.

一般的なシミュレーション技法およびディスク・ベースの多重プログラミング・システムのシミュレーションのためのモデルを議論している．この論文は計算機システムのシミュレーションに関する注釈付きの参考文献一覧表を含んでいる．

McIlroy, M. D. [1969]. “Mass Produced Software Components,” in *Software Engineering* (P. Naur and B. Randell, eds.), pp. 138–155. NATO Sci. Affairs Div., Brussels.

この非公式の論文はソフトウェア・システムに対する棚ざらしになっていないモジュールの可能性を論じ，そのようなモジュールの生産に関する疑問点を調べている．

McKeeman, W. M., Horning, J. J., and Wortman, D. B. [1970]. *A Compiler Generator.* Prentice-Hall, New Jersey.

XPLコンパイラ記述システムに基づいたコンパイラの記述のための入門的な教科書である．

McKellar, A. C., and Coffman, Jr., E. G. [1969]. “Organizing Matrices and Matrix Operations for Paged Memory Systems,” *CACM* **12**, No. 3 (Mar.), 153–165.

行列のページ割りつけを再配置することにより，著者等はページ化の振舞に劇的な改善を得た．

McKinney, J. M. [1969]. “A Survey of Analytical Time-Sharing Models,” *ACM Comput. Surveys* **1**, No. 2 (June), 105–116.

この論文はラウンド・ロビン，フォアグラウンド・バック・グラウンド，フ

ィードバック待ち行列などを含んだスケジューリング・アルゴリズムの待ち行列モデルを概括している．この論文は注釈付き参考文献一覧表で終っている．

Madnick, S. E., and Alsop, J. W. [1969]. "A Modular Approach To File System Design," *Proc. AFIPS* **34**, *SJCC*, pp. 1–14.

ファイル・システムの構成に対する多重レベルの方法を概説している．これは第6章で提示した構造の基礎となっている．

Marshland, T. A., and Tartar, J. [1973]. "A Course in Minicomputer Systems," *ACM SIGCSE Bull.* **5**, No. 1 (Feb.), 153–156.

ミニコンピュータに対するコースの概略を紹介している．コースの目標，プロジェクトの分野，適当なミニコンピュータの選択等を議論している．

Mattson, R., Gecsei, J., Slutz, D., and Traiger, I. [1970]. "Evaluation Techniques for Storage Hierarchies," *IBM Systems J.* **9**, No. 2, 78–117.

ページ化の環境に於ける記憶の階層を解析している．置き換えアルゴリズムの一つのクラス"スタック・アルゴリズム"の性質を議論し，一つのスタック・アルゴリズムが最適であることを証明している．

Mealy, G. H. [1967]. "Operating Systems (Excerpts)," in *Programming Systems and Languages* (S. Rosen, ed.), pp. 516–534. McGraw-Hill, New York.

これは入出力とスーパーバイザに集中して，IBM流のオペレーティング・システムを非常に一般的に論じたものである．

Mealy, G. H. [1969]. "The System Design Cycle," *Proc. Second ACM Symp. Operating Systems Principles* (Oct.), pp. 1–7.

この論文は組織上の困難さに関係するシステム開発の問題を総括している．Conway の方則を引用している．この内容は第8. 4 節に関係している．

Mealy, G. H., Witt, B. I., and Clark, W. A. [1966]. "The Functional Structure of OS/360," *IBM Systems J.* **5**, No. 1, 2–51.

IBM システム/360 に対する OS オペレーティング・システムの統一的な概

説を行なっている.

Mills, H. [**1971**]. "Top Down Programming in Large Systems," in *Debugging Techniques in Large Systems* (R. Rustin, ed.), pp. 41–55. Prentice-Hall, Englewood Cliffs, New Jersey.

構造的プログラミングの2つの基本的な概念すなわち連続的改良と単純な制御構造とをプログラム設計の全体的なすすめ方の一部として議論している.

Mills, H. [**1972**]. *Mathematical Foundations of Structured Programming*, N. FSC 72-6012. IBM Federal Systems Div.

この論文はどんなプログラムでも, IF-THEN-ELSE, BLOCK, DO-UNTIL の制御構造（すなわち GO TO なし）だけを用いて書くことができることを示すのが, 中心の話題となっている. これらの制御構造を用いて下降法で書いたプログラムの正当性の証明に対する基本的な述語も提示している.

Mills, H. [**1973**]. "On the Development of Large Reliable Programs," *Record 1973 IEEE Symp. Comput. Software Reliability, New York*, pp. 155–159.

構造的プログラミングと大規模で信頼性のあるソフトウェアの構成に関するそれらの効果との総合的見解を提示している.

Morenoff, E., and McLean, J. B. [**1967**]. "An Approach To Standardizing Computer Systems," *Proc. ACM Nat. Conf., 22nd*, pp. 527–536.

多くの施設における頻繁なハードウェアの変更を考えると, 機械の間のプログラムの互換性の問題は重要である. 著者等はこの問題を調べ, 究極的な標準化への方向をいくつか示している.

Morse, P. M. [**1958**]. *Queues, Inventories, and Maintenance*. Wiley. New York.

この教科書は待ち行列理論の概括をした初期のドキュメントの一つである.

Murphy, J. E. [**1968**]. "Resource Allocation With Interlock Detection In A Multi-Task System," *Proc. AFIPS* **33**, Pt. II, *FJCC*, pp. 1169–1176.

待ち行列管理の技法と資源割り当てに対する行列に基づいた管理法とを議論している．この方式を用いたデッドロックの検出アルゴリズムを提示している．

Naur, P., and Randell, B. (**eds.**) [1969]. *Software Engineering.* NATO Sci. Affairs Div., Brussels.

1968年度のソフトウェア工学に関する NATO 会議に対する論文，議論、ワークショップの概要を報告している．

Neumann, P. G. [1969]. "The Role Of Motherhood In The Pop Art Of System Programming," *Proc. Second ACM Symp. Operating Systems Principles* (Oct.), pp. 13–18.

システム設計と実現に対する常識的な原則を議論している．そのような原則を約1ダース示し，これらの原則が何故頻繁に無視されるのかを解いている．簡潔で，興味がわき，教えることの多い提示である．

Oppenheimer, G., and Weizer, N. [1968]. "Resource Management for a Medium Scale Time-Sharing Operating System," *CACM* **11**, No. 5 (May), 313–322.

RCA Spectra 70/46 時分割オペレーティング・システムの設計に関連してスケジューリング，資源割り当て，記憶管理の問題を議論している．

Organick, E. I. [1972]. *The Multics System: An Examination of its Structure.* MIT Press, Cambridge, Massachusetts.

MULTICS システムに関して，ばらばらにあった情報をこのような非常に読み易い読みものにまとめた．これは良く知られたオペレーティング・システムに関する出版物の中で良いものに属するであろう．

Organick, E. I. [1973]. *Computer System Organization—The B5700/B6700 Series.* Academic Press, New York.

この本は Burroughs の機械のソフトウェアの構造とハードウェアのアーキテクチャの密接な関連性を紹介している．計算機システムの構成に対する一元的進め方の一部としてセグメント化，プロセスの構造，割り込み，スタック命令などを議論している．

Pankhurst, R. J. [1968]. "Program Overlay Techniques," *CACM* 11, No. 2, (Feb.), 119-125.

オーバーレイ技法の自動的，半自動的，手動的なものを述べている．実現におけるいくつかの疑問点も考慮している．

Parnas, D. L. [1971]. "Information Distribution Aspects of Design Methodology," *Proc. IFIP Congr. 1971*, Booklet TA-3, pp. 26-30.

モジュール間の結合の度合はモジュールがお互にしなければならない仮定の数によって測ることができる．この前提のもとに，著者はソフトウェア・システムの設計とドキュメンテーションに対するその影響を議論している．

Parnas, D. L., and Darringer, J. A. [1967]. "SODAS And A Methodology For System Design," *Proc. AFIPS* 31, *FJCC*, pp. 449-474.

この論文はシステム設計とシミュレーション用言語 SODAS の立場から下降法的システム設計の方式を議論している．この論文は下降法的進め方の例を深く展開している．

Poole, P. C., and Waite, W. M. [1969]. "Machine Independent Software," *ACM Symp. Operating Systems Principles* (Oct.), 19-24.

抽象機械のモデル化とマクロ処理の技法とを機械に独立なソフトウェアを記述するシステムの開発に用いている．実現したシステムはソフトウェアの移動性に対するいくつかの応用において，有効であったことを示している．

Randell, B. [1968]. "Towards A Methodology of Computing System Design," in *Software Engineering* (P. Naur and B. Randell, eds.), pp. 204-208. NATO Sci. Affairs Div., Brussels.

ZurcherとRandell [1969] が下降法および上昇法的設計に対して開発した構造を用いて，Dijkstra の THE の設計方式と Parnas と Darringer の SODAS言語との比較をしている．

Randell, B., and Kuehner, C. J. [1968]. "Dynamic Storage Allocation Systems," *CACM* 11, No. 5 (May), 297-306.

この論文は記憶の動的割り当てに対する機構を概括している．7個のよく知

られたシステムで用いている番地付けの技法を紹介している.

Rice, D. E., and van Dam, A. [**1971**]. "An Introduction to Information Structures and Paging Considerations for Online Text Editing Systems," *Advances in Information System Sci.* **4,** 93–159.

著者等はテキスト・エディティング・システムにおける経験をページの大きさやページ化とセグメント化などの記憶の構成に対する疑問点の解明に適用している.

Rodriguez, J. E. [**1967**]. *A Graph Model For Parallel Computation,* MAC-TR-64. Ph. D. Thesis, Dept. of Elec. Engrng., M. I. T., Cambridge, Massachusetts (Sept.).

並列計算に対する"プログラム・グラフ"モデルを論じ, 解析している. 他のプログラム図式と比較して, このモデルの主要な特徴は制御カウンタが演算子節点の定義の中に陰に含まれていることである (第Ⅱ. 2節参照).

Rose, C. S. [**1972**]. "LOGOS and the Software Engineer," *Proc. AFIPS* **41,** Pt. I, *FJCC,* pp. 311–323.

ハードウェアとソフトウェアの設計に用いるソフトウェアの環境である LOGOSのプロジェクトの総括を行なっている. グラフ理論的なシステムの表現方法を詳しく紹介している.

Rosin, R. F. [**1969**]. "Supervisory And Monitor Systems," *ACM Compt. Surveys* **1,** No. 1 (Mar.), 37–54.

この論文はオペレーティング・システムの歴史的な概括である. システムの内部構造よりもそれが提供する機能に重点をおいている.

Saltzer, J. H. [**1966**]. *Traffic Control In A Multiplexed Computer System,* MAC-TR-30. Sc. D. Thesis, Dept. of Elec. Engrng. Cambridge, Massachusetts.

この博士論文は多重プログラミング, 多重プロセサ・システムにおける, プロセサの多重化の方式を述べている. 提案したトラフィック・コントロールは MULTICS システムで実現された.

Saltzer, J. H., and Gintell, J. W. [1970]. "The Instrumentation of Multics," *CACM* **13**, No. 8 (Aug.), 495–500.
特別なハードウェアの装置とソフトウェアの道具を MULTLCS システムの評価の際に使用したことを紹介している.

Sammet, J. E. [1971]. "A Brief Survey of Languages Used in Systems Implementation," *ACM SIGPLAN Notices* **6**, No. 9 (Oct.), 1–19.
オペレーティング・システムの実現において, 高級言語を選択するときの基本的問題を議論している. いくつかの特別な言語を紹介し, それらの言語でシステムの例を書いている.

Sayers, A. P. (ed.) [1971]. *The Comtre Corp. Operating Systems Survey.* Auerbach, Princeton, New Jersey.
オペレーティング・システムの基本概念と機能とを概括し, 14個の "実際の" オペレーティング・システムを紹介している.

Sayre, D. [1969]. "Is Automatic 'Folding' Of Programs Efficient Enough To Replace Manual ?" *CACM* **12**, No. 12 (Dec.), 656–660.
要求ページ化の性能を伝統的なオーバレイの性能と比較している. ページ化は費用と柔軟性とに関して利点を持つことを示している.

Seaman, P. H., Lind, R. A., and Wilson, T. L. [1966]. "An Analysis of Auxiliary Storage Activity," *IBM Systems J.* **5**, No. 3, 158–170.
ダイレクト・アクセス周辺記憶システムに対する待ち行列モデルを用いて, 装置の使用率, 応答時間, 待ち行列の長さの推定値を求めている.

Schroeder, M. D., and Saltzer, J. H. [1971]. "A Hardware Architecture For Implementing Protection Rings," *Proc. Third ACM Symp. Operating Systems Principles* (Oct.), pp. 42–54.
セグメント化した番地空間における保護リングの概念を, ハードウェアに対する考慮も幾分強調しながら論じている.

Sevcik, K. C. [1971]. *The Use Of Service Time Distributions In*

Scheduling. Ph. D. Thesis, Inst. For Comput. Res., Univ. of Chicago, Chicago, Illinois.

単一処理装置に対する最適およびほとんど最適なスケジューリング法則を理論的に解析している．いろいろと異った仮定に対して，法則を解析している．特に，損失を最小にし処理を最大にする“最小ランク”と呼ぶ最適法則を紹介している．

Sevcik, K. C., Atwood, J. W., Grushcow, M. S., Holt, R. C., Horning, J. J., and Tsichritzis, D. [**1972**]. “Project SUE as a Learning Experience,” *Proc. AFIPS* **41**, Pt. I, *FJCC* pp. 331–339.

SUEシステムの立場から，いろいろの設計のやり方を議論している．SUEシステムで導入した特別な機構の概略を述べている．

Sharpe, W. F. [**1969**]. *The Economics of Computers*. Columbia Univ. Press, New York.

この本は計算することの経済性に関連した経済学の基本的な題材からはじまっている．次に，計算機を取り入れて，それを使用するときの経済的な問題を詳細に論じている．

Shaw, A. C., and Weiderman, N. H. [**1971**]. “A Multiprogramming System for Education and Research,” *Proc. IFIP Congr. 1971*, Booklet TA-7, pp. 110–114.

著者等は小型のオペレーティング・システムに対するベースとして用いることのできるような高級言語でシミュレートした簡単な機械を紹介している．このシステムは教育と研究の両方の目的のために使用している．

Slutz, D. R. [**1968**]. *The Flow Graph Model Of Parallel Computation*, MAC-TR-53. Ph. D. Thesis, Dept. of Elec. Engrng., M. I. T., Cambridge, Massachusetts.

Karp と Miller の並列プログラム図式 [Karp および Miller, 1969] の修正版を決定性，等価保存変換の観点から解析している．

Snowdon, R. A. [**1972**]. “PEARL: An Interactive System for the Prepa-

ration and Validation of Structured Programs," *ACM SIGPLAN Notices* **7**, No. 3 (Mar.), 9–12.

この論文は下降的プログラミングの過程において，各レベルの改良の一貫性を検証するためにプログラマを助けるようなシステムを紹介している．

Teichrow, D., and Sayani, H. [**1971**]. "Automation of System Building," *Datamation* **17**, No. 16 (Aug. 15), 25–30.

この非公式な文献は Information System Design and Optimization System (ISDOS) プロジェクトの紹介である．プロジェクトの目標はシステム設計と実現の過程をできるだけ多く自動化することである．

Teorey, T. J., and Pinkerton, T. B. [**1972**]. "A Comparative Analysis Of Disk Scheduling Policies," *CACM* **15**, No. 3 (Mar.), 177–184.

解析的およびシミュレーションのモデルを用いて，可動ヘッドのディスクに対する5つのスケジューリング方式を評価している．これらの方式をシーク時間の平均値と待ち時間の平均値に関して比較している．全体で一番良い方法を選んでいる．

Thompson, K., and Ritchie, D. M. [**1973**]. "The Unix Time-Sharing System," *Fourth Symposium on Operating System Principles* (Oct.).†

この論文は DEC PDP-11/40 と 11/45 の機械に対する Unix Time-Sharing System の重要な特徴を紹介している．Unix の階層的ファイル・システムは特に第6章と関連がある．

Turnbull, C. J. M. [**1972**]. *A Comparative Analysis of Several Disk Scheduling Algorithms*, Tech. Rep. CSRG-16. Comput. Systems Res. Group, Univ. of Toronto, Toronto.

いくつかのディスク・スケジューリングの方式を解析している．結果は Teorey と Pinkerton [1972] によるシミュレーションのデータと一致している．

† 訳注．同じ論文は，CACM にものっている
CACM **17**, No. 7, 365—375 (July, 1974)

Van Horn, E. C. [1968]. "Three Criteria For Designing Computing Systems To Facilitate Debugging," *CACM* **11**, No. 5 (May), 360-365.

システムの特徴を評価する3つの判定基準をシステムの出力を制御するという立場から述べている．仮想機械の考え方を用いて，この判定基準を説明している．

Varney, R. C. [1971]. "Process Selection In A Hierarchical Operating System," *Proc. Third ACM Symp. Operating Systems Principles* (Oct.), pp. 106-108.

プロセス・スケジューリングのモデルを紹介し，RC 4000 のプロセス構造に適用している．

Weinberg, G. M. [1971]. *The Psychology of Computer Programming.* Van Nostrand-Reinhold, Princeton, New Jersey.

コンピュータ・プログラミングは，心理学的方法を用いて，人間の活動として研究しなければならないという挑戦的な議論をしている．

Weingarten, A. [1966]. "The Eschenbach Drum Scheme," *CACM* **9**, No. 7 (July), 509-512.

ページ化用ドラムの構成を紹介し解析している．

Weissman, L., and Stacey, G. M. [1973]. "An Interface System for Improving Reliability of Software Systems," *Record 1973 IEEE Symp. Comput. Software Reliability, New York*, pp. 136-142.

モジュールのインターフェイスをとるための自動的システムを紹介している．モジュールはシステムが物理的場所に写像する論理的名前を用いて連絡する．後者はモジュールが直接アクセスすることはできない．

Wilkes, M. V. [1965]. "Slave Memories And Dynamic Storage Allocation," *IEEE Trans. Computers* **14**, No. 4 (Apr.), 270-271.

この論文は，平均の記憶アクセス時間を高速化するための，小型のキャッシュ記憶に関する初期の文献である（Kuck および Lawrie [1970] 参照）．

Wilkes, M. V. [**1968**]. *Time Sharing Computer Systems.* Amer. Elsevier, New York.

この本は内部記憶とファイル構造に少し重きをおいた時分割システムをユーザから見た記述である.

Wilkes, M. V. [**1969**]. "A Model For Core Space Allocation In a Time-Sharing System," *Proc. AFIPS* **34,** *SJCC*, pp. 265-271.

少量の処理の要求しかないジョブのスワッピングをなくし，またできればいくらかのスワッピングのオーバーヘッドを除去するようなパイプラインを用いた記憶の構造を紹介している.

Winograd, J., Morganstein, S. J., and Herman, R. [**1971**]. "Simulation Studies of a Virtual Memory Time-Shared Demand Paging Operating System," *Proc. Third ACM Symp. Operating Systems Principles* (Oct.), pp. 149-155.

時分割オペレーティング・システムの詳細なシミュレーション・モデルを紹介している. 2つのシミュレーションの例に対し記憶の大きさ，ページ化の率，バッチと会話型のスケジューリングの偏差に関連して議論を行なっている.

Wirth, N. [**1968**]. "PL360—A Programming Language For the 360 Computers," *JACM* **15,** No. 1 (Jan.), 37-74.

この論文は IBM システム/360 の機械に対するシステム・プログラミング用言語 PL360 の完全な記述である.

Wirth, N. [**1969**]. "On Multiprogramming, Machine Coding And Computer Organization," *CACM* **12,** No. 9 (Sept.), 489-498.

割り込みを陽に用いる事を避けた，並列処理のためのプログラミング言語の構造を議論している. Dijkstra の腕木を PL360 のプログラムの例に適用している.

Wirth, N. [**1971**]. "The Programming Language Pascal," *Acta Informat.* **1,** Fasc. 1, 35-63.

構造的プログラミングの言語 PASCAL をこのわかり易い論文ですべて紹介している.

Wulf, W. A. [1969]. "Performance Monitors For Multiprogramming Systems," *Proc. Second ACM Symp. Operating Systems Principles* (Oct.), pp. 175-181.

この論文はスケジューリングの決定に影響を与えるために用いる情報を集めるような性能モニタの使用を議論している. 特に, 割り当て方針の調整とジョブ・ミックスの修正を議論している.

Wulf, W., Mitchell, J., Hopkins, M., Horning, J., Wells, M., Berglass, G., Halstead, M., and Karasz, P. [1972]. "Project Rosetta Stone—An Exhibition of Language for System Implementation—Participants Guide." Dept. of Comput. Sci. Carnegie-Mellon Univ., Pittsburgh, Pennsylvania (Sept.).

システム記述言語の比較をするロゼッタ・ストーン・プロジェクト（別名, "法典作り"）を詳細に記述している.

Zurcher, F. W., and Randell, B. [1969]. "Iterative Multi-Level Modeling—A Methodology For Computer System Design," *Proc. IFIP Congr. 1968*, pp. 867-871. North-Holland Publ. Amsterdam.

この古典的な論文はシステム設計に対する下降法的な進め方を提示している.

索　　引

あ　行

さ　行

ま　行

わ　行

《訳者略歴》

斎藤信男
さいとうのぶお

昭和15年　松本市に生まれる

昭和39年　東京大学工学部計数工学科卒業

昭和41年　東京大学大学院工学系研究科修士課程修了

昭和41年　通産省工業技術院電気試験所（現電子技術総合研究所）入所．電子計算機部計算機方式研究室，ソフトウェア部言語処理研究室などを歴任．その間タイムシェアリング・システムの研究開発，システム記述言語の研究開発などに従事．

昭和45年〜昭和46年　スタンフォード大学に留学(科学技術庁在外研究員)．プログラムの理論，オペレーティング・システムの理論などの研究を行なう．

昭和49年　筑波大学電子・情報工学系専任講師

昭和50年　筑波大学電子・情報工学系助教授

現　在：　慶応義塾大学工学部数理工学科助教授・工博

計算機ソフトウェアの基礎となるプログラムの理論，並列処理の理論，オペレーティング・システムの理論などの研究に従事．ACM，情報処理学会，電子通信学会，計測自動制御学会各会員．

オペレーティング・システムの基礎

1976年5月20日	初版印刷	©1976	日本コンピュータ協会
1976年6月10日	初版発行	著　者	Tsichritzis / Bernstein
1978年5月10日	2刷発行	訳　者	斎　藤　信　男
1982年2月15日	**3刷発行**	発行者	村　越　勝　弘
		発行所	**日本コンピュータ協会**
			東京都文京区本郷3丁目5-5
			電話 東京（03）813—6929番

検　印
廃　止

乱丁本・落丁本はお買求め書店にてお取りかえいたします　Printed in Japan.